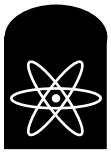
UCRL-ID-122526

A Proposed Acceptance Process for Commercial Off-the-Shelf (COTS) Software in Reactor Applications

Prepared by
G. G. Preckshot
J. A. Scott

Prepared for
U.S. Nuclear Regulatory Commission



FESSP
Fission Energy and Systems Safety Program

Lawrence Livermore National Laboratory

Disclaimer

This work was supported by the United States Nuclear Regulatory Commission under a Memorandum of Understanding with the United States Department of Energy, and performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

A Proposed Acceptance Process for Commercial Off-the-Shelf (COTS) Software in Reactor Applications

Manuscript date: September 22, 1995

Prepared by
G. G. Preckshot
J. A. Scott

Lawrence Livermore National Laboratory
7000 East Avenue
Livermore, CA 94550

Prepared for
U.S. Nuclear Regulatory Commission

ABSTRACT

This paper proposes a process for acceptance of commercial off-the-shelf (COTS) software products for use in reactor systems important to safety. An initial set of four criteria establishes COTS software product identification and its safety category. Based on safety category, three sets of additional criteria, graded in rigor, are applied to approve (or disapprove) the product. These criteria fall roughly into three areas: product assurance, verification of safety function and safety impact, and examination of usage experience of the COTS product in circumstances similar to the proposed application. A report addressing the testing of existing software is included as an appendix.

CONTENTS

TABLES

ACKNOWLEDGMENT

The authors thank and acknowledge the efforts of Nuclear Regulatory Commission staff members, Leo Beltracchi, Robert Brill, John Gallagher, Joe Joyce, Joel Kramer, and James Stewart, who reviewed this work and provided their insights and comments.

EXECUTIVE SUMMARY

The approval process for commercial off-the-shelf (COTS) software to be used in reactor safety systems (Class 1E) has been termed “commercial dedication,” although this term also implies defect reporting responsibilities (for the dedicator) under 10 CFR 21. Since this document addresses only the investigation of the acceptability of such software for use in systems important to safety, the term “acceptance process” is used. The purpose of this work is to review current and draft standards to create a set of “acceptance criteria” and incorporate them into a proposed acceptance process. The resulting acceptance criteria are assessed with regard to NRC practices and regulatory purview to arrive at an ordered set of criteria related to safety that comprises a proposed process for accepting COTS software for use in reactor safety applications. Prior to discussing the acceptance process, summary information is provided regarding the nature of the problem of acceptance and the feasibility of using COTS software in reactor safety applications. The latter describes some cost-related considerations, other than purchase price, that are associated with using COTS software in systems important to safety.

In keeping with NRC practices, wherein reactor equipment is regulated primarily in proportion to its importance to reactor safety, it is proposed that COTS products should be reviewed with a stringency proportional to the safety functions they are intended to provide. An initial set of four criteria, comprising the preliminary phase of the acceptance process, establishes COTS product identification and its safety category. Based on safety category, one of three sets of additional criteria, graded in rigor, is applied to approve (or disapprove) the product. These criteria fall roughly into three areas: product assurance, verification of safety function and safety impact, and examination of usage experience of the COTS product in circumstances similar to the proposed application.

Several conclusions are drawn. First, it is feasible to design an acceptance process based on a classification of software with respect to its importance to safety. Second, the rank order of acceptance criteria is dictated by data dependencies. The exercise of satisfying first-ranked criteria produces data that are necessary for the remaining criteria. Thus, no basis for satisfying subsequent criteria exists if “upstream” criteria are not satisfied. Finally, no single standard extant at this writing completely addresses the acceptance problem. Taken in combination, however, a usable set of criteria for determining the acceptability of a COTS software item can be derived from IEC, IEEE, and ISO standards. Based on the results, it appears that acceptable COTS software items can be produced by vendors who are generally aware of the risks associated with systems important to safety and who employ accepted software engineering practice to produce high-integrity software.

A PROPOSED ACCEPTANCE PROCESS FOR COMMERCIAL OFF-THE-SHELF (COTS) SOFTWARE IN REACTOR APPLICATIONS

1.0 INTRODUCTION

1.1 Scope

This report addresses the use of commercial off-the-shelf (COTS) software in those nuclear power plant (NPP) systems that have some relationship to safety. The report proposes a process for determining the acceptability of COTS software using a classification scheme based on the importance to safety of the system in which the COTS product will be used. Since software testing is related to the acceptance process, the report, *Testing Existing Software for Safety-Related Applications*, has been included as Appendix B of this report.

1.2 Purpose

The purpose of this report is to present a proposed acceptance process, based on a review of current and draft standards, for the use of COTS software items in NPP systems important to safety. The process is centered on suitable "acceptance criteria" that are supported by inclusion in a majority of standards publications or work-in-progress and that are consistent with NRC practices and regulatory purview.

1.3 Definitions

Key terms used in the report are defined below.

Class 1E

The safety classification of the electric equipment and systems that are essential to emergency reactor shutdown, containment isolation, reactor core cooling, and containment and reactor heat removal, or are otherwise essential in preventing significant release of radioactive material to the environment.

Commercial-grade item

A structure, system, or component, or part thereof that is used in the design of a nuclear power plant and which could affect the safety function of the plant, but was not designed and manufactured as a basic component.[1]

Commercial-grade dedication

An acceptance process undertaken to provide reasonable assurance that a commercial-grade item to be used as a basic component will perform its intended safety function and, in this respect, will be deemed equivalent to an item designed and manufactured under a 10 CFR Part 50, Appendix B, quality assurance program.[2]

Critical software

Software that is part of, or could affect, the safety function of a basic component or a commercial-grade software item that undergoes commercial-grade dedication.

Important to safety

A structure, system, or component:

a. whose failure could lead to a significant radiation hazard,

b. that prevents anticipated operational occurrences from leading to accident conditions, or

c. that is provided to mitigate consequences of failure of other structures, systems or components.

This encompasses both safety and safety-related systems.

Safety-related

Pertaining to systems important to safety but that are not safety systems.

---

[1] This definition is sufficient for this report; see 10 CFR Part 21 (in the revision process as of this writing) for the complete current definition.

[2] Since this report is concerned with the specifics of gaining reasonable assurance, and not with the aspects of 10 CFR that will apply following the actual acceptance of a commercial-grade item, the process of gaining assurance is called an "acceptance process" rather than a "commercial dedication process."

Safety systems

Those systems that are relied upon to remain functional during and following design basis events to ensure (a) the integrity of the reactor coolant pressure boundary, (b) the capability to shut down the reactor and maintain it in a safe shutdown condition, or (c) the capability to prevent or mitigate the consequences of accidents that could result in potential offsite exposures comparable to the 10 CFR Part 100 guidelines.

Statistical certainty

An assertion made within calculated confidence limits supported by data samples and an underlying distribution theory.

Statistical validity

An assertion is statistically valid if the attributes of the data supporting the statistical certainty of the assertion are consistent with the inference to be made.

(The term, statistical validity, is used in several places in this report to refer to operating experience of a commercial-grade software item. The connotations of this usage are:

- for each datum in the operating experience database, the version and release numbers of the involved software item are identified and match the target commercial-grade software item; and,
- for each datum, the operating environment, configuration, and usage are reported and match the intended environment, configuration, and usage of the target commercial-grade software item; and,
- all received reports and incident details are recorded in the database, regardless of initial diagnosis; and,
- an estimate[3] is made of the expected number of unreported, unique incidents,[4] with confidence limits; and,
- the number of reports in the database, the confidence interval, and the expected number of unreported severe errors are consistent with the intended use of the commercial-grade software item.)

[3]An estimate can be made from the frequency distribution of reports of unique incidents, or direct sampling of the software user data source.

[4]A unique incident is one that, after root-cause analysis, has a different root cause from all previously reported incidents.

1.4 Background

Considerable interest exists in the nuclear power community in the potential use of commercial off-the-shelf software in nuclear power plant systems. For safety-related systems, it is necessary to evaluate the acceptability of the COTS software for use in the system and then to formally designate the COTS software as a "basic component" of a system essential to reactor safety. This is referred to as "dedication of a commercial-grade item" by 10 CFR Part 21, although the term "commercial dedication" was sometimes also used to signify only the formal acceptance of the product and assumption of 10 CFR 21 defect-reporting responsibilities. Since this report addresses only the evaluation and acceptance of COTS software in both safety-related systems and in other systems important to safety, the process is referred to herein as an "acceptance process."

The acceptance process used to determine the acceptability of a COTS software item is currently the subject of much debate. This section of the report discusses key issues related to the feasibility of using COTS software in systems important to safety, a brief discussion of the varying perspectives of typical participants, and background information regarding the development of the proposed acceptance process. The following sections of this report address the classification of software to be used in NPP systems, discuss how various standards influenced the proposed acceptance process, and describe the proposed acceptance process itself.

1.4.1 COTS Background and Feasibility

1.4.1.1 Commercial-Off-the-Shelf Software and its Acceptability

COTS software has the potential to yield large cost savings if it can be used in safety systems and other systems important to safety in nuclear power plants. The COTS software of interest typically includes compilers, operating systems, software supplied in Programmable Logic Controllers (PLCs), and software in commercial industrial digital control systems. The problem faced by the nuclear reactor industry is to show that a particular COTS product, which may be useful in a nuclear reactor instrumentation and control system, has sufficient reliability for the application. The best solution to the problem is that the software engineering group that produced the product did its work using the necessary processes for producing high-quality software, and that evidence of this (including documentation, tests, inspections, quality assurance/control, verification and validation, and various other quality-related activities) is available for inspection by the prospective buyer. Lacking this favorable situation, some minimum standards by which a COTS product is judged should be available. The

central issue in establishing these minimum standards is that the COTS product must be shown to have sufficient quality for its intended application. A fundamental concern for regulators in approving an acceptance process is that if the process is significantly less rigorous than normal regulatory review of software developed in-house, it may become a conduit for escaping necessary scrutiny.

To date, this process has been rather informal. Recently, a number of standards committees have been addressing the problem of formalizing the process. Various techniques have been proposed for dealing with specific technical problems frequently encountered in applying acceptance processes; however, considerable controversy still surrounds many of these.

1.4.1.2 Feasibility Issues

The primary motivation for considering the use of COTS software as an alternative to a new software development is to avoid unnecessary development costs. Although the cost savings appear obvious at first glance, there are important issues affecting costs that require careful consideration. Many of these issues relate to the fact that the potential COTS software product must be demonstrated to be of sufficient quality for its intended application in a system important to safety.

One such issue is the existence, availability, and relevance of information needed to demonstrate quality. Discussions regarding the demonstration of confidence in COTS software products (Gallagher, 1994) indicate that there are basically three potential sources for pertinent information: an examination of the development process and the associated product documentation, testing of the COTS software product, and an examination of the operational history associated with the product. The workshop participants speculate that information from these sources might be used in varying mixtures depending on context; but they provide no details on how this might be done while ensuring that the appropriate quality is demonstrated. There is a danger that such alternatives could be used to avoid the scrutiny attached to new software development efforts.

Relevant standards indicate that information from all three sources is needed and that possibilities are limited regarding recourse to one source when information is not available from another. The core information is provided by product documentation, records, and details of the development process applied to the product. Testing of a COTS software product can be used for several purposes, including

- augmenting the testing effort conducted during development
- addressing requirements specific to the proposed application of the COTS item in a system important to safety
- verifying of the intended functions of the product, and
- assessing the quality of testing activities carried out during development (see Appendix B of this report for information on testing with an emphasis on COTS products).

Operational history provides supplementary information that can complement testing. The draft supplement to IEC 880 states that “for safety critical (Category A) systems, the feedback of experience should never prevent a careful analysis of the product itself, of its architecture, of how it has been developed and of the validation of all the functionalities intended to be used in the design.” IEEE-7-4.3.2-1993, in its discussion of qualification of existing commercial computers, states that “exceptions to the development steps required by this standard or referenced documents may be taken as long as there are other compensating factors that serve to provide equivalent results.” For an information source to provide “equivalent results,” the subject of the compensation must be tightly focused on a particular technical question, and it must be shown how the compensating information is equivalent to the missing information. For safety-essential systems, the necessary demonstration of quality will require extensive information about the product and a rigorous analysis of that information. Options for dealing with missing information are limited and require careful consideration and documentation. The activities required for demonstrating quality may be quite costly.

Another consideration associated with demonstrating confidence in a COTS software product is the potential impact of functions contained in the COTS software that are not required for its proposed application, such as undocumented functions or unused resident functions (called “unintended functions” and “unused functions” in this report) The commitment to use COTS software requires that the potential impact of unintended functions and inadvertent actuation of unused functions be assessed in the process of determining acceptability. The activities required to make this assessment can represent significant additional costs.

In addition to the costs described above, the problems associated with maintaining and tracking COTS software status should be considered carefully, especially defect reporting as detailed in 10 CFR Part 21. The downstream costs associated with effecting this maintenance and tracking capability may be comparable to those associated with software developed directly for the NPP environment. This area

includes configuration management and the vendor's long-term role, obsolescence and the potential cost of system redesign, bug tracking and reporting commitments, and the implementation and requalification of bug fixes.

1.4.1.3 Perspectives on Acceptability Evaluations

New software developed for the NPP environment can be controlled from inception to address a wide variety of assurance and safety considerations. This is not the case for COTS software, which has already been developed, and whose developers may be responsive to a number of commercial objectives unrelated to NPP safety. In particular, the COTS product is unlikely to have been the result of development processes specifically attuned to safety and hazards analyses of NPPs.

This is economically important because COTS software products may supply needed functions where it is impractical to implement those functions by developing new software. For this approach to be safe, questions that need to answered are

1. What assurance and safety considerations should be addressed?
2. Is the COTS item fully consistent with those considerations?

The rigor of these questions is affected by the relative importance to safety of the proposed COTS software item. The first issue to address in evaluating acceptability is importance grading (classification) with respect to safety role. Given the proposed safety classification of an identified COTS item, the next problem is verification of its properties and quality.

If the COTS item was produced by a vendor with systematic and well-controlled software processes, many of the documentary products necessary to make the product and process determinations will exist and be verifiable, and therefore determination of properties and quality would be fairly straightforward. If the COTS item was produced in a less mature development environment, the issue is complicated by the fact that quality assurance processes may not have been employed, or may have been employed in an inconsistent fashion. In this case, the COTS item performance of its functions is suspect, and assurance investigations to address this question are hampered by the lack of—or poor quality of—the associated materials that would have been generated by a mature software process.

The problems of identification of safety role and verification of properties and quality are complicated by the fact that there are three perspectives on the evaluation of acceptability:

- the producer of the COTS item, i.e., the COTS software vendor
- the user (customer) of the COTS item, i.e., a reactor vendor or an owner/operator doing a retrofit
- the regulator responsible for approving the use of the COTS item. The regulator has the legal responsibility of certifying that the NPP in which the COTS item will be used is safe.

The effect of these perspectives can be demonstrated by considering various scenarios for the evaluation of a particular COTS product.

Scenario 1: *A COTS software vendor wants to dedicate a product for specific uses.*

In this case, the COTS software vendor would be directly concerned with regulator needs and user needs during the acceptance process, implying a cooperative COTS software vendor that probably has relatively mature software development processes. The vendor is motivated by business advantage, possibly with respect to meeting similar standards in other fields (e.g., environmental), and takes responsibility for generic, but not specific, safety analyses.

Scenario 2: *A COTS software user—for example, a reactor vendor—dedicates a COTS product for use in a reactor design.*

The COTS software vendor may not be strongly motivated, in which case a good reactor vendor relationship with COTS software vendor would be instrumental in the acceptance process. The reactor vendor would be responsible for coordinating activities with the COTS software vendor and with the regulator, and for specific safety analyses.

Scenario 3*: A COTS software user—for example, an owner/operator—dedicates a COTS item for use in a retrofit.*

An owner/operator will have a long-standing relationship with the regulator, but perhaps not with respect to software development issues. An owner/operator is somewhat removed from the original reactor vendor and may not have the same understanding of the design subtleties of reactor systems important to safety. The owner/operator would probably use the reactor vendor's existing thermal/hydraulic safety analyses, but would be responsible for determining the COTS product safety functions. An owner/operator may also be more removed from the COTS software vendor than the reactor vendor.

Scenario 4: *A regulator permits use of a previously qualified COTS item for a certain class of service.*

This scenario would be a generalization of an existing qualification of the COTS item by an applicant. The regulator would need to have high confidence in the COTS item. There would be possible standardization benefits, but these would depend upon the acceptability to the regulator of safety analyses regarding class of service, as opposed to plant-specific analyses. Dedication for generic class of service would not absolve the designer using the COTS item from performing specific, use-related safety analyses.

1.4.2 Background on the Proposed Acceptance Process

1.4.2.1 Classification

While it is necessary to demonstrate that a COTS item has sufficient reliability for its intended application, it is also important that the demonstration be commensurate with the importance to safety of the COTS item. That is, the acceptance process must ensure sufficient quality but should not require unnecessary effort. Just as reactor subsystems and equipment are regulated primarily in proportion to their importance to reactor safety, COTS products should be reviewed with a stringency proportional to the safety function they are intended to provide. This allows regulatory resources to be applied efficiently and does not burden reactor vendors with unnecessary requirements.

1.4.2.2 Basis for the Acceptance Criteria

Current Standards

Standards for software quality assurance (SQA), software configuration management (SCM), software verification and validation (SVV), and software criteria for use in nuclear power plants were reviewed for criteria appropriate to COTS products. In many cases, no explicit provision is made for adapting existing software to a critical application; the standards assume that such software will be developed as new software products. There are provisions for qualifying software products for use in producing the final product, but in most cases, these provisions amount to ensuring that the standard itself was employed by the software subcontractor. The following standards were reviewed to determine criteria either explicitly required for COTS products or implicitly required because the COTS product was required to conform to the standard:

- IEEE 730 (Now 730.1), “IEEE Standard for Software Quality Assurance Plans”
- IEEE 983 (P730.2, Draft 4), “IEEE Guide for Software Quality Assurance Planning”
- IEEE 828, “IEEE Standard for Software Configuration Management Plans”
- IEEE 1042, “IEEE Guide to Software Configuration Management”
- IEEE 7-4.3.2, “Standard Criteria for Digital Computers in Safety Systems of Nuclear Power Generating Stations”
- ISO 9000-3, “Guidelines for the Application of ISO 9000-1 to the Development, Supply, and Maintenance of Software”
- ANSI/ANS-10.4, “Guidelines for the Verification and Validation of Scientific and Engineering Computer Programs for the Nuclear Industry”
- ANSI/IEEE 1012, “IEEE Standard for Software Verification and Validation Plans”
- IEC 880, “Software for Computers in the Safety Systems of Nuclear Power Stations”
- IEC 987, “Programmed Digital Computers Important to Safety for Nuclear Power Stations”

An overview of the pertinent aspects of each of the listed standards is given in Section 3, and a detailed multi-tabular list of criteria abstracted from the standards may be found in Appendix A.

New Standards Activity

New work is being performed on acceptance criteria for COTS products by the IEC, driven by the potential economic advantage of being able to use existing software products. A draft addition to IEC 880 was used to review the criteria extracted from existing standards for completeness and applicability. This is discussed in overview in Section 3.10. IEC 1226 provides a *de facto* safety categorization, which is discussed in detail in Section 2.1. The following emerging or new standards were reviewed:

- First Supplement to IEC 880 (Draft), “Software for Computers in the Safety Systems of Nuclear Power Stations”
- IEC 1226, “The Classification of Instrumentation and Control Systems Important to Safety for Nuclear Power Plants.”

Design Factors

Previous work on vendor assessment (Lawrence & Preckshot, 1994, Lawrence et al., 1994) was applied to check the reasonableness of the COTS assessment criteria derived as described above. It became clear that the design factors primarily address product assurance issues, which for COTS products is only part of the problem. The vendor assessment work also provides

the approach and rationale for judging the COTS assessment criteria against NRC needs.

NRC Review

A preliminary version of this report was presented to the NRC. The comments received at that meeting have been incorporated into this version of the report.

Expert Peer Review Meeting on High-Integrity Software

This meeting was conducted by Mitre Corporation for the NRC Office of Nuclear Regulatory Research, and substantial discussions on COTS issues ensued. Material from the NRC was provided by an NRC representative. Excerpts from these discussions were analyzed and considered in completing this version of the report.

1.4.2.3 Acceptance Process, Criteria, and Conclusions

While it is not possible to completely eliminate subjectivity and the consequent variability of results, the acceptance process presented in Section 4 has been developed to a sufficient level of detail to promote reasonable uniformity of results on each key element. The process consists of preliminary activities that apply regardless of safety category, followed by a set of activities tailored to the particular safety category established for the COTS item in its intended usage. A set of ranked criteria is listed for three safety categories based on review of the criteria listed in Appendix A. The acceptance process is compatible with IEEE-7-4.3.2-1993, with detail supplied from other standards in places where IEEE-7-4.3.2 requires “engineering judgment.” This level of stringency is consistent with the body of IEEE-7-4.3.2, which addresses software development in general. The systems-oriented approach of the IEC standards has had a significant influence on the resulting list of acceptance criteria, adding a risk assessment step that the other standards lack. An interesting and possibly surprising conclusion is that the rank order is the result of simple data dependencies. The achievement of a particular criterion is dependent upon satisfaction of preceding criteria, so that from a practical viewpoint, the importance of individual criteria cannot be decided in isolation.

2.0 SAFETY CATEGORIES

Safety categorization fulfills its intended purpose if sufficient categories exist to enable efficient application of regulatory resources, but not so many that efforts are fragmented. The appropriate number appears to be more than two (safety and non-safety) and less than five. The categorization problem has three parts. The first is to define categories, for which this paper has recourse to IEC 1226. The second is to deduce to which category a COTS product belongs, which is discussed below. The third part is to decide what rigor of acceptance process is appropriate to each category, which is considered in Section 4.

2.1 IEC 1226 Categories

IEC 1226 proposes, by implication, four categories—A, B, C, and unclassified—which in this context means "has no safety impact." Rather than repeat IEC 1226 definitions, Table 1 shows by example some familiar reactor systems and where they would be placed in the IEC 1226 scheme (IEC 1226 2/6/93, Annex A). IEC 1226 category A is very similar to IEEE Class 1E. An approximate equivalence to Regulatory Guide 1.97 signal categories is also shown.

2.2 COTS Usage Categories

Unfortunately, many COTS products do not fit neatly into IEC 1226 categories. This is because COTS products, although there may be extant examples of category A, B, or C usage, are also used in supporting roles that may affect software in categories A, B, or C. Table 2 below summarizes the possibilities.

Table 3 formalizes the decision process detailed above. The operative principles are that if an error in COTS software can occur in operation important to safety or can embed an error in other software important to safety, then the COTS software takes on the category of the software in which the error can occur. If the COTS software can only challenge software important to safety, possibly exposing existing errors, then the COTS software takes on the next lower safety category. Since category C has relatively low reliability requirements, software that produces category C software may be of standard commercial quality (unclassified).

2.3 Special Note on Compilers, Linkers, and Operating Systems

Compilers, linkers, operating systems used for development, and similar COTS software are among those COTS products that have potentials for embedding errors in software that is essential or important to safety, but are not themselves executing when the error causes a challenge. Most standards are silent or say very little about qualifying such software, because the dilemma is a difficult one to resolve. In general, there is a trade-off; is it safer to use or not to use such a product? If the answer were a simple "don't use it," safety software would still be written in machine language, an obvious absurdity. Even with the success of modern software tools, however, trusting acceptance of such tools is not recommended. Tools should be rated for safety impact as detailed in Tables 1–3, and assurance methods used for similar tools used for hazardous applications should be applied.

For example, the draft supplement to IEC 880 notes that it can be quite difficult to demonstrate that a compiler works correctly. The draft supplement states that "Even validated compilers have been found to contain serious errors." This was illustrated by the experience with Ada compilers; there was a considerable delay before qualified Ada compilers became generally available. An "Ada qualification suite" of programs that an Ada compiler should successfully compile or detect errors in now has hundreds of thousands of programs and is still growing as compiler writers discover newer and subtler ways to introduce bugs in Ada compilers.

Because of the difficulties associated validating compilers, linkers, and operating systems, the evaluation should be based on best available information and should be continuous while the tool is in use. Where qualification tests exist (such as the Ada qualification suite), only products that pass such tests should be accepted. In addition, extensive statistically valid operational experience is important in these cases because the validation effort is beyond the skills of most unspecialized software developers. Sometimes this may mean using a product version that has a known bug list as opposed to the latest version on the market. There may be less risk in using an older version and avoiding well-known bugs than in using the latest version with a high expected level of unreported, severe errors. These considerations also apply, to a lesser extent, in the category B and category C processes described in Sections 4.3 and 4.4 below.

Table 1. Safety Categories

IEC 1226 Category	Example Systems	RG 1.97 Equivalent Category
A	Reactor Protection System (RPS) Engineered Safety Features Actuation System (ESFAS) Instrumentation essential for operator action	A,B A,B A,B,C,D
B	Reactor automatic control system Control room data processing system Fire suppression system Refueling system interlocks and circuits	 E
C	Alarms, annunciators Radwaste and area monitoring Access control system Emergency communications system	B,C,D,E C,E

Table 2. COTS Usage Categories

Usage Category	Description	Equivalent IEC 1226
Direct	Directly used in an A, B, or C application.	A, B, or C
Indirect	Directly produces executable modules that are used in A, B, or C applications (software tools such as compilers, linkers, automatic configuration managers, or the like).	
	Produces A modules	A or B[5]
	Produces B modules	B or C[6]
	Produces C modules	unclassified
Support	CASE systems, or other support systems that indirectly assist in the production of A, B, or C applications, or software that runs as an independent background surveillance system of A, B, or C applications.	unclassified
Unrelated	Software that has no impact on A, B, or C applications.	unclassified

Table 3. COTS Safety Category Criteria

1	If the COTS product is used directly in a system important to safety, the COTS safety category is determined by the criteria of IEC 1226.
2	If the COTS product directly produces or controls the configuration of an executable software product that is used in a system important to safety, and no method exists to validate the output of the COTS product, the COTS safety category is the same as that of its output, except that category C software may be produced by COTS products of the unclassified category. COTS software that directly produces category A or B software that is validated by other means is category B or C, respectively.
3	If the COTS product supports production of category A, B, or C software, but does not directly produce or control the configuration of such software modules, it falls under the safety category "unclassified."
4	If the COTS product has no impact on category A, B, or C software or systems, it falls under the safety category "unclassified."

[5]The choice of A or B category depends upon whether the A module has diverse alternatives or whether there is another software tool, treated as category A, that verifies the output of the subject tool.

[6] The choice of B or C category depends upon whether the B module has diverse alternatives or whether there is another software tool, treated as category B, that verifies the output of the subject tool.

3.0 OVERVIEW OF STANDARDS REVIEWED

If there is a general philosophical difference between standards, it may be the tendency to take a pro forma approach versus the tendency to be prescriptive. Predominantly pro forma standards, such as IEEE and ISO software standards, require developers to produce documents and perform certain activities, but do not prescribe many details or pass/fail criteria. Abstracting criteria from such standards requires judgment and understanding of the underlying software production and validation processes, an endeavor that may be subject to differing opinions. Standards that tend to be prescriptive, of which the three IEC standards are examples, are more detailed and leave less to professional judgment, although they do not eliminate the potential for differing viewpoints. A detailed standard may lose current applicability, requiring professional judgment to apply its strictures to evolving technology. In the following, our estimate of the approach taken in each standard is mentioned.

3.1 IEEE 730 (Now 730.1)

IEEE 730.1 is a pro forma standard that describes the activities and documentation required for software quality assurance (SQA) plans. By implication, this standard addresses only two formal categories of software (critical and non-critical). Some or all of the standard may be applied to non-critical software, but the degree of application is optional. The standard acts as an umbrella standard in the sense that it requires some sort of software configuration management (SCM) and some sort of software verification and validation (SVV). Other IEEE standards on SCM and SVV are referenced by this standard. Table A-3[7] lists the activities and documentation required, which are presumed to extend to safety-critical COTS products by Table A-3, entry 9 and Table A-9, entry 1.

3.2 IEEE 983 (P730.2, Draft 5)

This standard is a guidance standard for applying IEEE 730.1. As such, it does not supersede the requirements of that standard or impose additional requirements. It provides clarification, as in Table A-3, entry 6, and all entries in Table A-11.

3.3 IEEE 828

IEEE 828 presents pro forma requirements for activities and documentation in software configuration management plans for all criticality levels of software; the standard makes no distinction between levels.

Table A-4 lists the detailed requirements for configuration management plans. Entry 8 in this table lists the crucial points with regard to configuration management maintained by a supplier. IEEE 828 requires a description of how acquired software will be received, tested, and placed under SCM; how changes to the supplier's software are to be processed, and whether and how the supplier will participate in the project's change management process. IEEE 828 does not address COTS software explicitly, or specify criteria that software configuration management systems of a COTS software vendor should meet.

3.4 IEEE 1042

IEEE 1042 provides guidance by example for applying IEEE 828. As a guidance standard, this document does not contradict or add to the requirements stipulated by IEEE 828.

3.5 ISO 9000-3

The ISO 9000 standards apply to quality assurance programs in general, and are not limited to software. ISO 9000-3 interprets the general standards as applied to software, and fulfills somewhat the same role as IEEE 730.1; that is, it is a pro forma standard that acts, in part, as an umbrella standard, mentioning other aspects of software quality such as SCM and SVV. The ISO standards are more contractually oriented than the IEEE standards, and somewhat more generally written as far as criteria for standard adherence are concerned. Tables A-3, line 9, Table A-4, line 8, and Table A-9, line 1 reflect the ISO view of subcontracted or existing software products.

3.6 ANSI/ANS-10.4

This standard regards verification and validation of scientific and engineering programs for use in the nuclear industry, and typical programs used for simulation or design of reactors or reactor subsystems. It is the only standard, of all reviewed, that considers the question of verification and validation of existing computer programs for which there is little or no documentation. This probably reflects the actual situation extant with this type of software; little or no formal software engineering method is applied during software development, leaving a software product of unknown reliability. ANSI/ANS-10.4 suggested many of the entries in Tables A-6, A-12, and A-13, and was useful in expanding the functional requirements of ANSI/IEEE 1012.

[7] Tables marked with an A- may be found in Appendix A.

3.7 ANSI/IEEE 1012

This is a pro forma standard that describes the activities and documentation required for verification and validation of critical software. An example of the difference between the pro forma and prescriptive approach can be seen in Table A-12, wherein ANSI/ANS-10.4 is used to expand the minimum V&V tasks specified by IEEE 1012 with criteria for performance. V&V tasks are construed to apply to COTS products by virtue of the requirements in IEEE 730.1, as expressed in Table A-9. ANSI/IEEE 1012 is summarized in Table A-5 and auxiliary tables that expand detailed V&V requirements.

3.8 IEC 880

IEC 880 is a prescriptive standard which offers detailed criteria that software under its purview must satisfy. The relatively poor organization of this standard may detract from its effectiveness, but it is consistently better than the IEEE standards in its "systems" approach. Risk-related requirements are emphasized, as are interfaces with and relations to other systems and hardware, which differs significantly from the IEEE and ISO standards. The following five-point summation of Section 5 of IEC 880 illustrates the risk-based approach:

- Safety relevance of software parts should be determined;
- More limiting recommendations apply to risky parts;
- High-safety-impact software modules should be easily identifiable from system structure and data layout;
- Available testing and validation procedures should be considered when doing the design;
- If difficulties arise, a retrospective change of style may be required.

"Self supervision" is required, meaning that the software includes code to detect hardware errors and errors committed by software. Self supervision is only regarded in the literature as effective for detecting hardware errors; considerable controversy still exists on whether effective means exist to detect software errors with more software.

3.9 IEC 987

IEC 987 is a systems and hardware prescriptive standard that defers to IEC 880 on specific software issues. The "systems" slant of IEC 880 is discussed above.

3.10 IEC 880, First Supplement to IEC 880 (Draft)

IEC 880 provided a strong connection between risks or safety and software (or system) requirements, and this connection is continued and enhanced in the draft supplement. This document places strong emphasis on determining the safety functions that a COTS product will perform before deciding on the rigor of the acceptance process to be followed. This is combined with a strict view of experience data; for important safety functions, COTS experience data must be relevant and statistically valid. The draft addition had a significant effect on the review of candidate acceptance criteria compiled from the IEEE and ISO standards. With the exception of entry 3, all other entries in Table 4, below, were motivated by the IEC 880 supplement. Likewise, items 7–9 of Table 5, and items 7 and 8 of Tables 6 and 7 can be specifically attributed to IEC 880's strong requirement for risk coverage. (These tables may be found in Section 4.) The IEC 880 supplement also had particular criteria for judging experience databases, and this is reflected in entries 10 and 11 of Table 5, and entries 9 and 10 of Tables 6 and 7.

3.11 IEEE-7-4.3.2-1993

While the proposed acceptance process presented in this report draws heavily on IEC 880, it is also generally consistent with IEEE-7-4.3.2-1993. This standard addresses testing for COTS items as well as consideration of software development methods and operating experience. The standard has a subjective nature, however, as evidenced by the following:

> "Exceptions to the development steps required by this standard or referenced documents may be taken as long as there are other compensating factors that serve to provide equivalent results."
>
> "Acceptance shall be based upon an engineering judgment that the available evidence provides adequate confidence that the existing commercial computer, including hardware, software, firmware, and interfaces, can perform its intended functions."

While the general intent of these passages is clear, there is room for a varying strictness of interpretation. In interpreting these passages with respect to the acceptance process proposed in this report, it was assumed that it must be explicitly and convincingly shown how information from a compensating factor provides equivalent results and, when engineering judgment is used, that it be applied to specific, narrowly defined questions and that its basis be convincing and documented. This standard was

reviewed in this context for possible omissions in the candidate list of COTS acceptance criteria.

3.12 IEC 1226

IEC 1226 provides the missing link that the other standards discussed herein lack: a consistent definition of safety categories. This standard uses terms familiar to those involved in nuclear power plant safety: redundancy, diversity, defense-in-depth, and reliability. While other choices of safety category could be made, the categories in this standard are credible and usable.

4.0 PROPOSED ACCEPTANCE PROCESS

The proposed acceptance process is based on the classification scheme described in Section 2 and on a set of acceptance criteria derived from the standards described in Section 3. It is broken into two phases: a preliminary qualification phase, and a detailed qualification phase. The preliminary qualification phase applies to all COTS products, regardless of the ultimate safety categorization. This phase is concerned with understanding system safety requirements, understanding the COTS product's proposed role in a system important to safety, unambiguously identifying the COTS product, and determining the rigor of subsequent qualification procedures. The detailed qualification phase activities vary in rigor and content depending upon the result of the preliminary phase. Successful completion of the appropriate detailed phase qualifies (pending formal acceptance/dedication) the COTS item for the specific intended use that was analyzed and documented in the preliminary phase.

The proposed COTS acceptance criteria are presented in Table 4 in dependency order in tabular form. A short discussion of each criterion and the reason for dependency on previous criteria or why subsequent criteria are dependent follows. As with an earlier assessment of software design factors (Lawrence & Preckshot 1994), COTS acceptance criteria were reviewed for potential effect of each criterion, observability, and pertinence to NRC practices and procedures. The product quality of greatest pertinence to NRC concerns is the product's potential safety impact or safety category. For this reason, safety category determines differences in the rigor of the acceptance criteria. The criteria presented below are organized into four tables, with the latter three corresponding to acceptance process requirements specific to each of the three safety categories. The first table corresponds to the preliminary phase of the process and directs the reviewer to the applicable table of the latter three. In a number of cases, recourse is taken to the Appendix for detailed requirements. This does not imply that these requirements are less important, but only that the level of detail may obscure the instant discussion.

4.1 Commercial-Grade Dedication for Class-of-Service

When a COTS item is accepted for a generic class of service, a distinction must be made between the responsibilities of the dedicator and the designer who applies the product to a specific safety application. The dedicator is responsible for generic safety issues, such as defining the service class, the criteria for deciding if a particular application falls within that service class, defect reporting responsibilities that must be assumed by the prospective user, and the design verification techniques that must be used by the designer applying the generic COTS item to a particular safety application. The commercial dedication process verifies that the COTS item is of sufficient quality and has the required functions to meet class-of-service functional requirements. Equally important, the dedicator's review of product software requirements and software quality assurance provides confidence that unintended functions are unlikely and that reliable means exist to prevent the activation of unused functions.

Commercial-grade dedication for a generic class-of-service cannot absolve the application designer of the responsibility for making a safety case for specific applications of the dedicated COTS item. In this respect, COTS software is no different than a dedicated commercial-grade hardware item, such as a relay; the product received must still be shown to be the product specified, and the design using the item or the method of application must still be shown to be correct and consistent with the terms of the dedication under design control and quality assurance measures required by 10 CFR Part 50, Appendix B.

Table 4. Preliminary COTS Acceptance Criteria

1	Risk and hazards analyses shall be used to identify system-level safety functions required.
2	The safety functions (if any) that the COTS product will perform shall be identified.
3	The COTS product shall be under configuration and change control. See Table A-4 for detailed SCM criteria.
4	The safety category of the COTS product shall be determined. Proceed to Table 5, 6, or 7 depending upon category A, B, or C, respectively.

4.2 Preliminary Phase of the Proposed Acceptance Process

The preliminary criteria should be applied to all COTS products, recognizing that some of these criteria (criterion 1, for instance) will likely be reviewed for other reasons. The ranking of these criteria (developed below) is determined by data dependencies; that is, satisfaction of earlier-ranked criteria (lower number rank) produces information that is required to determine if later-ranked criteria are satisfied.

4.2.1 Acceptance Criterion 1—Risk and Hazards Analyses

System-level risk and hazard analyses must be complete, as they provide the basis for determining the required system safety functions, some of which may be performed by the COTS item under review. For generic class-of-service dedications, the system-level risk and hazard analyses must define the plant and safety environment in which the generic COTS item is expected to function. Since this analysis is the foundation upon which a safety determination is made about COTS item usage, an incomplete analysis or incomplete review of existing analyses may result in an unreviewed safety question. Typically, such system-level analyses are done for nuclear reactors as part of the licensing process, but the analyses may require updating to accommodate plant modifications in existing plants.

By implication, all of the IEEE and ISO standards assume that the risk category is already known. The IEC standards make the requirement for understanding risks explicit.

Rationale for ranking:

Risk and hazards analyses were taken as the criterion required before any COTS product can be considered because, if the system risks and hazards are unknown, it is not possible to determine what risks and hazards are incurred by introducing a COTS product.

4.2.2 Acceptance Criterion 2—Identification of Safety Functions

Once the system risks are known, determining how the COTS product will fit into a risk management scheme is next. The intended use of the COTS item should be completely described and documented, all the safety functions of the COTS item should be fully described, and the intended relationship of the COTS item to other systems essential or important to safety should be clearly stated. Any omitted usage, function, or relationship is construed to be unintended, and may result in an unreviewed safety question. A COTS item is acceptable only for usage and functions that are documented during the acceptance evaluation. A COTS item that is being dedicated for generic class-of-service is acceptable only for service within the functional and performance limits established in this step. This does not relieve an engineer applying a generic class-of-service COTS item of the responsibility for making a safety case for the particular functions the COTS item will perform; the generic dedication only supplies an acceptable way of performing those functions provided terms and conditions of the dedication are met.

IEC 880 makes this process explicit as “identifying the safety functions” of the software product, whether it is COTS or to-be-developed software. IEEE-7-4.3.2-1993 refers to this criterion as “identifying the safety functions the computer must perform.”

Rationale for ranking:

This step is not possible until the system-level risks and hazards have been analyzed.

4.2.3 Acceptance Criterion 3—Configuration Management

A mechanism for software configuration management must exist, and the COTS product under review must be clearly identified and under management control as a configuration item. If a COTS product falls within regulatory purview, regardless of potential safety categorization, it should be identified as a configuration item and be under configuration management control, either by the COTS supplier, the owner/operator, or the reactor system vendor. For COTS products in nuclear reactor systems essential or important to safety, the rigor of configuration management should be independent of safety category. The goal at this point in the process is to ensure that the COTS product in question is a mature product that has been completely and clearly identified to all parties in the process. The configuration identification cannot be a “moving target.” The configuration management system will be important in later steps because of ancillary items such as documentation and testing materials, status reporting mechanisms, problem reporting, change control, and release mechanisms.

Rationale for ranking:

Configuration management is ranked third because not only do most standards and the design factors mention this as a crucial criterion (Lawrence & Preckshot, 1994), but because a poorly identified and uncontrolled COTS product does not meet the intent of Criterion VIII, “Identification and Control of Materials, Parts, and Components,” of Appendix B, “Quality Assurance Criteria for Nuclear Power Plants and Fuel Reprocessing Plants,” of 10 CFR Part 50. The COTS product that is installed must be the same COTS product that was accepted.

4.2.4 Acceptance Criterion 4—Determination of Safety Category

The safety category of the COTS item in its intended use, as evaluated in Acceptance Criterion 2, should be determined according to IEC 1226 using the guidance given in Section 2. This determines the rigor of the remaining criteria.

Rationale for ranking:

The product cannot be placed in a safety category until the COTS product and its safety functions have been identified.

4.3 Detailed Acceptance Criteria for Category A

Detailed acceptance criteria for category A software is listed below in Table 5.

4.3.1 Acceptance Criterion A5—Product Assurance

For this category, the applicable standards require COTS products to be developed to the same rigor that would have been required were the product produced as a new software development for the intended safety application. A COTS product that was not developed under a plan that included software requirements, a software design, coding to internal or external standards, testing, V&V, and quality assurance audits would not be acceptable. An assessment of the COTS software vendor's development, validation and verification, and quality assurance processes should be made. Ideally, the COTS software vendor will make available the internal documents that can prove this. At a minimum, for this software safety category, COTS vendor development, testing, V&V, and quality assurance policies and procedures should be documented and the documents should be available. This is an appropriate place to apply the design factors described in Lawrence & Preckshot 1994, as a validity check on this assessment.

Table 5. Category A COTS Acceptance Criteria

A5	The COTS product shall have been developed under a rigorous Software Quality Assurance Plan as defined by IEEE 730.1, ISO 9000-3, or IEC 880. This shall include full V&V. See Table A-3 for detailed SQA criteria. See Table A-5 for detailed V&V criteria. See Table A-12 for minimum required V&V tasks.
A6	Documentation shall be available for review that demonstrates both Criterion A5 and that good software engineering practices were used, as detailed in Table A-7. Evidence shall be available that the minimum required reviews of Table A-8 were conducted.
A7	It shall be demonstrated that the COTS product meets the requirements identified in Criterion 2 (Table 4).
A8	It shall be demonstrated that the COTS product does not violate system safety requirements or constraints.
A9	The interfaces between the COTS product and other systems or software shall be identified, clearly defined, and under configuration management.
A10	The COTS product shall have significant (greater than 1 year) operating time,[8] with severe-error-free operating experience. At least two independent operating locations shall have used a product of identical version, release, and operating platform encompassing the same or nearly the same usage as the proposed usage. Any adverse reports, regardless of operating location, shall be considered. The configuration of the products in the experience data base shall closely match that of the proposed COTS product.[9]
A11	All errors, severe or otherwise, shall be reported to and analyzed by the COTS supplier. Procedures and incentives shall be in place to ensure a high level of demonstrated compliance, or the COTS supplier shall demonstrate with statistical certainty[10] that the error reporting system achieves this compliance. An error tracking, documentation, and resolution procedure shall document each error from report to resolution.
A12	Additional validation and testing shall be performed if needed to compensate for a small amount of missing documentation or alterations in configuration.

[8] Measured as in-service execution time concurrently at two or more customer sites.

[9] See the definition of statistical validity in Section 1.3.

[10] See the definition of statistical certainty in Section 1.3.

Satisfaction of this acceptance criterion by a generic class-of-service COTS item does not absolve the user of such an item of the responsibility for quality assurance measures in the application of the item. For example, a programmable logic controller (PLC) must be programmed in a ladder-logic or other programming language. Users of such devices would still be responsible for a 10 CFR Part 50, Appendix B quality assurance program, or whatever quality assurance programs were required by their license basis, applied to the design work the user does to incorporate the class-of-service COTS item in basic components.

Rationale for ranking:

Product assurance activities are ranked fifth in importance because this is the first time that the rigor required, the system safety requirements, and the COTS product safety requirements are all known.

4.3.2 Acceptance Criterion A6—Product Documentation

The reality of COTS products is that documentation is likely to be sparse and the COTS software dedicator may have difficulty gaining access to proprietary information related to software development. Nevertheless, sufficient documentation must exist to support the activities of following acceptance criteria, i.e., the satisfactory performance of these activities must not be prevented by missing documentation, such as missing source code. At a minimum, product documentation should include quality assurance certification that the COTS product[11] has met the vendor's own criteria identified in step A5—complete product user documentation that describes in detail how to apply and use the product, known bug lists, and error recovery procedures. Availability of source code is preferable; however, source code is not included in this minimum documentation unless questions associated with the other acceptance criteria can only be reasonably answered with approaches that include analyses or testing based on the source code. For example, questions about the adequacy of testing or V&V procedures examined in step A5, or questions raised based on the examination of operating experience and error reporting in steps A10 and A11, might indicate the need for additional static analyses or structural tests. The demonstration in step A8 to confirm that unintended functions will not impair system safety or questions about interfaces raised in step A9 could also indicate a need for static analyses.

The product documentation should describe all of the attributes identified in step 2 as necessary for performance of the safety functions assigned to the product. No undocumented feature can be used to perform a safety function, or is acceptable for this purpose. The user documentation should be testable; that is, product operation should be described unambiguously so that testing could determine if the product were defective. Additional testing to establish confidence in the product may be necessary. Information on specific considerations for testing COTS software can be found in Scott and Lawrence, 1995 (included as Appendix B). A product that does not match the performance specifications in product documentation is unacceptable.

Rationale for ranking:

Product documentation goes hand-in-hand with product assurance and is a necessary item for the evaluation of product and system safety.

4.3.3 Acceptance Criterion A7—Product Safety Requirements

Assuming that product assurance and documentation give confidence in knowledge of the COTS product's attributes, then it is appropriate to ask if these attributes satisfy the safety functions expected of the product.

Rationale for ranking:

Product safety requirements are ranked seventh because product attributes cannot be known with reasonable certainty without product assurance and sufficient detail without product documentation.

4.3.4 Acceptance Criterion A8—System Safety

Other attributes or qualities of the COTS product should not impair system safety. COTS products, because they must be commercially viable, often have functions or options beyond those required to satisfy the identified safety functions of the previous criterion. They may also have undocumented functions, or "bugs." These are the unused and the unintended function problems, respectively, and they may be more severe with COTS products because of extra functions or configurations these products may have.

For unintended or unused functions, the role of the dedicator, whether for generic class-of-service usage or use in a specific basic component, is the same. Confidence that unintended functions are unlikely is obtained through applying Criteria A5 and A6. It must be possible for a designer to prevent inadvertent activation of unused functions so that unused functions cannot be activated by unauthorized personnel or foreseeable operator errors.

Additional system-level requirements fall upon the dedicator for class-of-service. Criteria for when defense-in-depth or diversity may be required must be established. These criteria describe the allowable fraction or enumerations of safety functions that may be entrusted to the generic class-of-service COTS item,

[11] Identified by exact version and release designation.

after which defense-in-depth and diversity considerations may require a different approach.

Rationale for ranking:

Just as product assurance was a necessary prerequisite for determining if a COTS product satisfies its required safety functions, it is also a prerequisite for investigating whether other known attributes or options could defeat system safety goals. This is also not possible without detailed product information available in product documentation.

4.3.5 Acceptance Criterion A9—Interface Requirements

Due to the requirement on category A subsystems for single-failure robustness, the interfaces between category A COTS products and other systems must be known and investigated.

Rationale for ranking:

Interface requirements are ninth in sequence and importance, all previous criteria being prerequisites.

4.3.6 Acceptance Criterion A10—Experience Database

Category A products require the most rigorous and statistically valid[12] experience data. These data must be for the same version of the COTS product in the same or nearly the same environment and usage.

Rationale for ranking:

If any of the previous criteria are violated, the COTS product is inappropriate for the application envisioned and encouraging reports of good performance are irrelevant. Consequently, product experience is ranked tenth.

4.3.7 Acceptance Criterion A11—Error Reporting Requirement

The choice of a COTS product only begins its odyssey as part of a system important to safety. In the Operations & Maintenance phase of the software life cycle, complete information on errors must be made available so that evaluations can be made and appropriate subsequent actions taken. This information must be maintained since the severity of past errors may be determinable only in retrospect. While the COTS software vendor is not responsible for error reporting under 10 CFR Part 21, the existence of vendor-supported defect databases is a positive factor.

Rationale for ranking:

The error-reporting requirement follows experience database in rank since future error reports may lead to a retrospective re-evaluation of some reports in the experience database.

4.3.8 Acceptance Criterion A12—Additional V&V Requirement

If, after reviewing a COTS product with respect to the previous criteria, some questions remain unanswered, additional validation may be required for the application in question.

Rationale for ranking:

The additional V&V requirement is ranked last since all previous criteria must have been satisfied to reach this conclusion.

4.4 Detailed Acceptance Criteria for Category B

Detailed acceptance criteria for category B software is listed below in Table 6.

4.4.1 Acceptance Criterion B5—Product Assurance

A subset of the rigorous category A product assurance activities is appropriate for category B COTS products. The COTS software vendor should have documented policies and procedures in place that meet the requirements stated in Table 6, item B5. Interface analysis of category A products or systems has already limited the extent to which category B products can affect category A systems.

4.4.2 Acceptance Criterion B6—Product Documentation

Provision of appropriate documentation will facilitate the appraisal process, but recourse to design factors is acceptable to a greater extent than with category A products. Note that this still requires justification.

4.4.3 Acceptance Criterion B7—Product Safety Requirements

Category B safety requirements typically consist of an operator assistance function and automatic control that prevents excursions into operating regimes that require safety functions provided by category A systems. In the U.S., the NRC also permits category B systems to back up category A systems in the event of rare common-mode failures of those systems. As with category A COTS products, product assurance and documentation are necessary before product functions are known with sufficient certainty to determine if the COTS product fulfills its expected safety functions.

[12] See the definition of statistical validity.

Table 6. Category B COTS Acceptance Criteria

B5	The COTS product shall have been developed under a quality assurance plan and a systematic software development process. See Table A-3, entries 5 through 10 for SQA criteria. See Table A-5, entries 3 through 7 for V&V criteria.
B6	Documentation shall demonstrate Criterion B5. See Table A-7 for minimum required documentation.
B7	It shall be demonstrated that the COTS product will fulfill its safety functions as identified in Criterion 2 (Table 4), and that its reliability is sufficiently high that it does not present a high frequency of challenges to category A systems.
B8	The COTS product shall be consistent with system safety requirements.
B9	The COTS product shall have operated satisfactorily in similar applications. The version and release of reported experience may not be identical to the proposed COTS product, but a consistent configuration management program and well-managed update program provide traceability and change control.
B10	Error reporting, tracking, and resolution shall be consistent and correctly attributable to version and release, and procedures and incentives are in place that ensure demonstrated compliance during the first year after a version is released. The version and release proposed have no major unresolved problems. A current bug list shall be available to COTS purchasers as a support option.

4.4.4 Acceptance Criterion B8—System Safety

Equipment and software of category B is allowed more latitude so that it can achieve significantly greater function. Consequently, the COTS product should be consistent with system safety requirements. This means that the product may not necessarily take a safe action during a system excursion, but it should not cause a system excursion when operating as specified. Without product assurance, this cannot be determined with sufficient certainty.

4.4.5 Acceptance Criterion B9—Experience Database

If the foregoing criteria are violated, the COTS product is inappropriate for the intended application and operational experience is irrelevant. Provided that the previous criteria are satisfied, relaxed statistical validity requirements, such as variations in usage, environment, configuration, and confidence limits, are acceptable. These relaxations should be justified based on expected increase in risk.

4.4.6 Acceptance Criterion B10—Error Reporting Requirement

The error reporting requirements, which are relaxed from category A, are appropriate for good-quality, well-supported, commercial-grade software products. Typically, such products experience a significant reduction in error reports after the initial period of free software support service terminates.

4.5 Detailed Acceptance Criteria for Category C

Detailed acceptance criteria for category C software is listed below in Table 7.

4.5.1 Acceptance Criterion C5—Product Assurance

Product assurance activities are limited to determining that good software engineering practices were followed and that crucial V&V was performed. The term “good software engineering practice” is used to mean that standards for software development are used systematically, that configuration management is effectively employed, and that software development practices are defined, documented, and implemented. It must encompass the documentation and V&V referred to by Table 7, C5.

4.5.2 Acceptance Criterion C6—Product Documentation

The required documentation is limited, and missing documentation may be reconstructed or compensated in part by design factor assessment. Product documentation, while not required to be complete, should be consistent with the intended application. The product documentation should at least describe product features, and it may cover several versions of the product.

4.5.3 Acceptance Criterion C7—Product Safety Requirements

The ability of the COTS product to perform its (limited) safety functions should be demonstrated. In

view of the possibly limited product documentation, testing may be required to demonstrate this criterion. Information on specific considerations for testing COTS software can be found in Scott and Lawrence, 1995 (included as Appendix B).

4.5.4 Acceptance Criterion C8—System Safety

The lack of adverse effect on and coordination with other system safety functions should be demonstrated. Since this demonstration depends upon knowing product attributes, product assurance to the extent that attributes are known is a prerequisite.

4.5.5 Acceptance Criterion C9—Experience Database

Experience with product operation is irrelevant unless the previous criteria are satisfied. Relaxed reliability constraints allow reliable operation in the proposed application to serve as an experience base, although other applications may have experienced difficulties.

4.5.6 Acceptance Criterion C10—Error Reporting Requirement

Error reporting requirements, since they concern the future, are ranked last as an acceptance criterion. An error reporting scheme managed by the dedicator and covering only applications known to the dedicator is sufficient for this category.

Table 7. Category C COTS Acceptance Criteria

C5	The COTS product shall have been developed according to good software engineering practices. Minimum documentation, such as in Table A-13, shall be available or reconstructable. Minimum V&V tasks, as in Table A-12, entries 2, 4, 8, 9, and 19–22, shall have been performed.
C6	Minimum documentation described in Criterion C5, including V&V task documentation, shall be available for inspection.
C7	The COTS product may enhance safety by improving surveillance, improving operators' grasp of plant conditions, assisting in maintenance activities, reducing demands on category A or B systems, monitoring or reducing the effects of radiation releases, or similar purposes. The product's performance of its intended effect shall be verified.
C8	It shall be demonstrated that the COTS product cannot adversely affect the safety functions of category A or B systems or software and that it will not seriously mislead operators.
C9	The COTS product must be shown to operate without serious malfunction in the instant application.
C10	An error reporting scheme shall be planned or in place that tracks malfunctions of this COTS product in applications controlled by this applicant. Documentation and records retention allow error histories of 5 years or length of service, whichever is shorter.

5.0 CONCLUSIONS

Based on guidance provided by IEC 1226, it is possible to classify software for use in nuclear power plants. Using this classification and guidance from current standards, an acceptance process for COTS software items can be defined to a reasonable level of detail. This process is based on a preliminary set of criteria applying to all classifications, coupled with a detailed set of criteria that are relaxed as the importance to safety of the COTS software item decreases.

Acceptance criteria for COTS products are easily ranked by the dependence of some criteria on the information produced by meeting other criteria. COTS acceptance criteria fall into rank order because of the data dependencies mentioned in earlier discussion. This rank ordering is not necessarily the same as would be used by a software developer to select a COTS product; rather, it represents an order in which a regulatory agency would expect a safety basis to be constructed.

Review of standards from multiple sources reveals that IEC standards provide the risk-based approach and extra detail on which the pro forma IEEE and ISO standards are implicitly based, but never address directly. Apparently, diversity is a useful concept even when applied to standards activities, as no single group of standards was adequate to address the COTS acceptability problem.

Based on the analyses supporting this report, it appears that the use of COTS software in the safety systems of nuclear power plants will be limited to well-defined conditions and to COTS software products for which the acceptability of the product can be clearly established. Acceptability can be established through a combination of (1) examination of the product and records of its development indicating that a complete and rigorous software engineering process was applied, (2) sufficient evidence of satisfactory operational experience and error reporting history, (3) additional testing, and (4) vendor assessment as necessary. The development of such COTS software items will probably require developer knowledge that the product will be used in systems with medium to high risks, as well as the use of software processes that have been designed to produce high-integrity software. Such software developers will be generally aware of the types of hazards associated with the systems in which their products will be used, and those hazards will have been considered in their designs.

If generic, class-of-service commercial-grade item dedications are possible under the Commission's regulations, the dedicator is responsible for resolving generic acceptability questions, setting criteria for application of the dedicated item, resolving defect reporting responsibilities, and defining acceptable design and design verification methods for the application of the item to specific nuclear power plant safety problems. The designer applying such a class-of-service item is still responsible for resolving specific safety questions, using the item within the terms and conditions of the dedication, and performing such work under the requirements of 10 CFR Part 50, Appendix B, or the applicable licensing basis.

REFERENCES

Gallagher, John, (ed.), "Discussions Related to COTS Obtained from Expert Peer Review Meeting on High Integrity Software for MPPs Conducted by MITRE for NRC Research." May 24–26, 1994.

Lawrence, J. Dennis, Warren L. Persons, G. Gary Preckshot, and John Gallagher, "Evaluating Software for Safety Systems in Nuclear Power Plants." Submitted to 9th Annual Conference on Computer Assurance, Gaithersburg, MD, June 27–30, 1994. UCRL-JC-116038, Rev. 1, Lawrence Livermore National Laboratory, 1994.

Lawrence, J. D., and G. G. Preckshot, "Design Factors for Safety-Critical Software." Lawrence Livermore National Laboratory, NUREG/CR-6294, December 1994.

Scott, J. A., and J. D. Lawrence, "Testing Existing Software for Safety-Related Applications," Lawrence Livermore National Laboratory, UCRL-ID-117224, September 1995.

U.S. Nuclear Regulatory Commission, "Instrumentation for Light-Water-Cooled Nuclear Power Plants to Assess Plant and Environs Conditions During and Following an Accident." In Regulatory Guide 1.97, Rev. 3, May 1983.

Other Applicable Documents:

"Guidelines for the Application of ISO 9000-1 to the Development, Supply, and Maintenance of Software." International Organization for Standardization (ISO), ISO 9000-3, 1987.

"Guidelines for the Verification and Validation of Scientific and Engineering Computer Programs for the Nuclear Industry." ANSI/ANS-10.4, May 13, 1987.

"IEEE Standard for Software Quality Assurance Plans." IEEE 730 (Now 730.1), August 17, 1989.

"IEEE Guide for Software Quality Assurance Planning." IEEE 983 (Draft 730.2), Draft 4, October 1992.

"IEEE Standard for Software Configuration Management Plans." IEEE 828, June 23, 1983.

"IEEE Guide to Software Configuration Management." IEEE 1042, September 10, 1987.

"IEEE Standard for Software Verification and Validation Plans." ANSI/IEEE 1012, September 18, 1986.

"IEEE Standard Criteria for Digital Computers in Safety Systems of Nuclear Power Generating Stations." IEEE-7-4.3.2-1993.

"Programmed Digital Computers Important to Safety for Nuclear Power Stations." IEC 987, First Edition, 1989.

"Software for Computers in the Safety Systems of Nuclear Power Stations." IEC 880,[13] First Edition, 1986.

Software for Computers in the Safety Systems of Nuclear Power Stations, First Supplement to IEC 880 (Draft), Draft supplied by member of SC45A.

"The Classification of Instrumentation and Control Systems Important to Safety for Nuclear Power Plants." IEC 1226, February 6, 1993.

[13]COTS products are called pre-existing software products (PESPs) in IEC publications.

APPENDIX A—PRELIMINARY LIST OF FACTORS

A preliminary list of acceptance criteria for COTS products was identified from

- IEEE 730 (now 730.1) SQA Plans
- IEEE 983 (now 730.2 draft) SQA Plan Guidance
- IEEE 828 Software Configuration Management Plans
- IEEE 1042 SCM Plan Guidance
- ISO 9000 as applied to software.

Subsequently, the limited scope of these standards was widened by including

- ANSI/ANS-10.4-1987—Guidelines for the verification and validation of scientific and engineering computer programs for the Nuclear Industry
- ANSI/IEEE 1012-1986—Software Verification & Validation Plans.

With the exception of ANSI/ANS-10.4-1987, none of these standards takes significant note of existing software. Consequently, appropriate acceptance criteria can only be inferred from those requirements stated for software developed under purview of the standards. A general requirement present in each standard—that software be developed under the aegis of that particular standard—is impractical for most COTS products. It would be a happy finding indeed to discover well-done documentation and complete records ready for review.

A set of potential COTS acceptance criteria, or at least subjects to investigate, are listed in the following tables. The tables are organized from the general to the particular. The general table points to particular tables of additional criteria to be investigated if the general criterion is true.

Table A-1. Failure Consequence Criteria

1	Are consequences of failure unacceptable?	See Table A-2
2	Are consequences of failure acceptable?	Terminate

Table A-2. Plan Existence Criteria

1	An SQA plan and documentation exist	See Table A-3
2	A configuration management plan exists	See Table A-4
3	A software V&V plan exists	See Table A-5
4	Some of the above do not exist	See Table A-6

Table A-3. SQA Criteria

1	Does the SQA plan cover the minimum required subjects in the required format?	Format and subject matter is standard-dependent, but most standards specify similar approaches See IEEE 730.1
2	Does the plan describe responsibilities, authority, and relations between SQA units and software development units?	IEEE 730.1
3	Is minimum documentation available?	See Table A-7 for required documentation. See Table A-10 for optional documentation.
4	Were the minimum SQA reviews and audits performed?	See Table A-8 for minimum required reviews and audits
5	Are standards, practices, conventions, and metrics that were used, described?	See Table A-11 for suggested areas of standardization
6	Were procedures for problem reporting, tracking, and resolving described?	IEEE 730.1
	Problems documented & not forgotten	IEEE P730.2
	Problem reports validated	IEEE P730.2
	Feedback to developer & user	IEEE P730.2
	Data collected for metrics & SQA	IEEE P730.2
7	Were configuration management practices followed?	See Table A-4
8	Were V&V tasks performed?	See Table A-5
9	Did other software suppliers contribute to the product?	See Table A-9. “The supplier is responsible for the validation of subcontracted work.” ISO 9000-3
10	What records were generated, maintained, and retained?	IEEE 730.1
11	What methods or procedures were used to identify, assess, monitor, and control risk during development of the COTS product?	IEEE 730.1

Table A-4. Software Configuration Management Criteria

1	Does the configuration management plan cover the minimum required subjects in the required format?	Format and subject matter is standard-dependent, but most standards specify similar approaches. See IEEE 828
2	Does the plan describe responsibilities, authority, and relations between configuration management units and software development units?	IEEE 828
3	At least one configuration control board (CCB) is required. Does the plan describe the duties and responsibilities of the CCB and relations between the CCB, SQA, and software developers? e.g., Authority & responsibility Role Personnel How appointed Relation of developers & users	IEEE 828
4	Does the configuration management operation provide the following required functions? Configuration ID (baselines) Configuration control Configuration status accounting & reporting Configuration audits & reviews	IEEE 828
5	Configuration management is founded upon the establishment of "configuration baselines" for each version of each product. Is each product or version uniquely identified and "baselined"?	IEEE 828
6	Is the level of authority required for change (i.e., change control) described? Appropriate subjects include: Change approval routing lists Library control Access control R/w protection Member protection Member identification Archive maintenance Change history Disaster recovery Authority of each CCB over listed configuration items	IEEE 828
7	Does status accounting include Data collection Identified reports Problem investigation authority Maintaining and reporting Status of specifications Status of changes Status of product versions Status of software updates Status of client-furnished items	IEEE 828

Table A-4. Software Configuration Management Criteria (cont.)

8	Are suppliers of software products (e.g., COTS) under control? For each supplier. . . Is the SCM capability known? How is SCM performance monitored? For each product... Is the version in use archived? Is the version ID'd & baselined? Is the product under change control? Are product interfaces under control? Are suppliers CM audits "visible"? Is there valid problem tracking? Regarding supplier records. . . What records are kept? Can reviewers obtain access to them? How good are they? What security does the supplier have?	IEEE 828 and IEEE 1042. ISO 9000-3
9	Are the records to be maintained identified and are there retention periods specified for each type of record?	IEEE 828
10	What additional policies and directives govern the configuration management?	See Table A-14 for a list of typical policies and directives.

Table A-5. Software V&V Criteria

1	Does the V&V plan cover the minimum required subjects in the required format?	Format and subject matter is standard-dependent, but most standards specify similar approaches. See IEEE 1012
2	Is the organizational structure of the V&V function described, including the independence (or lack thereof) of the V&V organization from the software development organization?	IEEE 1012
3	Have the minimum required V&V tasks been accomplished?	See Table A-12 for minimum tasks
4	Does the V&V function detect errors as early in the development process as possible?	IEEE 1012
5	Can software changes and their consequences be assessed quickly?	IEEE 1012
6	Are V&V functions coordinated with the software development life cycle?	IEEE 1012
7	Are significant portions of V&V data missing?	See Table A-6

Table A-6. Actions to Take When Data is Missing

1	Can missing data be reconstructed from other available data?	Reconstruct data (see Table A-13) and proceed to Table A-5. ANSI/ANS-10.4
2	Can missing data be reverse-engineered from existing software products?	Reverse-engineer data (see Table A-13) and proceed to Table A-5. ANSI/ANS-10.4
3	Is recovered data and/or usage experience and configuration control insufficient to justify intended usage?	See Table A-13 for minimum data. If insufficient, terminate with prejudice. ANSI/ANS-10.4 and IEEE 828
4	Is sufficient test data available to support intended usage?	Reconstruct tests and proceed to Table A-5. ANSI/ANS-10.4

Table A-7. Minimum SQA Documentation

1	Software Quality Assurance Plan	IEEE 730.1
2	Software Requirements Specification	IEEE 730.1
3	Software Design Description	IEEE 730.1
4	Software V&V Plan	IEEE 730.1
5	Software V&V Report	IEEE 730.1
6	User Documentation (Manuals)	IEEE 730.1
7	Software Configuration Management Plan	IEEE 730.1

Table A-8. Minimum Required SQA Reviews and Audits

1	Software Requirements Review	IEEE 730.1
2	Preliminary Design Review	IEEE 730.1
3	Conceptual Design Review	IEEE 730.1
4	Software V&V Plan Review	IEEE 730.1
5	Functional Audits (e.g., validations)	IEEE 730.1
6	Physical Audits (e.g., physical deliverables)	IEEE 730.1
7	In-Process Audits (e.g., life cycle stage verification audits)	IEEE 730.1
8	Managerial Reviews	IEEE 730.1

Table A-9. SQA, SCM, and V&V for Other Software Suppliers

1	SQA for a purchased product shall meet the same requirements as if it were developed in-house. For to-be-developed COTS, the other software supplier shall perform the requirements of IEEE 730.1. For previously developed COTS, the "methods used to assure the suitability of the product for (its intended) use" shall be described. "Software suppliers" shall select subcontractors on the basis of their ability to meet subcontract requirements, including quality requirements.	IEEE 730.1 ISO 9000-3
2	SCM for a purchased product shall meet the same requirements as if it were developed in-house. As a minimum, the other software supplier is required to implement the provisions of IEEE 828.	IEEE 828. See also Table A-4, line 8
3	V&V for COTS is not addressed, except indirectly through IEEE 730.1 through its provision requiring IEEE 730.1 compliance of the software supplier, or through ANSI/ANS-10.4 through its provisions for reconstruction of missing data.	See Table A-3, line 8, and Table A-6

Table A-10. Suggested Additional Documentation

1	Software Development Plan	IEEE 730.1
2	Standards & Procedures Manual	IEEE 730.1
3	Software Project Management Plan	IEEE 730.1
4	Software Maintenance Manual	IEEE 730.1
5	User Requirements Specification	IEEE 730.1
6	External Interfaces Specification	IEEE 730.1
7	Internal Interfaces Specification	IEEE 730.1
8	Operations Manual	IEEE 730.1
9	Installation Manual	IEEE 730.1
10	Training Manual	IEEE 730.1
11	Training Plan (for SQA personnel)	IEEE 730.1
12	Software Metrics Plan	IEEE 730.1
13	Software Security Plan	IEEE 730.1

Table A-11. Suggested Areas of Standardization

1	Documentation Standards	IEEE P730.2
2	Logical Structure Standards	IEEE P730.2
3	Coding Standards	IEEE P730.2
4	Comment Standards	IEEE P730.2
5	Testing Standards	IEEE P730.2
6	SQA Product & Process Metrics	IEEE P730.2

Table A-12. Minimum V&V Tasks

1	SVVP	IEEE 730.1 and IEEE 1012
2	Requirements (e.g., SRS) Analysis	IEEE 1012
	Existence	ANSI/ANS-10.4
	Clarity	ANSI/ANS-10.4
	Consistency	ANSI/ANS-10.4
	Completeness	ANSI/ANS-10.4
	All functions included	
	Environment specified	
	Inputs & outputs specified	
	Standards used specified	
	Correctness	ANSI/ANS-10.4
	Feasibility	ANSI/ANS-10.4
	Testability	ANSI/ANS-10.4
3	SRS Traceability Analysis	IEEE 1012 & ANSI/ANS-10.4
4	Interface Requirements Analysis	IEEE 1012 & ANSI/ANS-10.4
5	Test Plan Generation	IEEE 1012 & ANSI/ANS-10.4
6	Acceptance Test Plan Generation	IEEE 1012
7	Design Analysis	IEEE 1012
	Completeness	ANSI/ANS-10.4
	Correctness	ANSI/ANS-10.4
	Consistency	ANSI/ANS-10.4
	Clearness	ANSI/ANS-10.4
	Feasibility	ANSI/ANS-10.4
8	Design Traceability Analysis	IEEE 1012 & ANSI/ANS-10.4
9	Interface Design Analysis	IEEE 1012
10	Unit Test Plan Generation	IEEE 1012 & ANSI/ANS-10.4
11	Integration Test Plan Generation	IEEE 1012 & ANSI/ANS-10.4
12	Test Designs	IEEE 1012
	Code test drivers	ANSI/ANS-10.4
13	Source Code Analysis	IEEE 1012
	Conformance to standards	ANSI/ANS-10.4
	Adequate comments	ANSI/ANS-10.4
	Clear and understandable	ANSI/ANS-10.4
	Consistent with design	ANSI/ANS-10.4
	Strong typing	ANSI/ANS-10.4
	Error-checking	ANSI/ANS-10.4
14	Source Code Traceability	IEEE 1012
15	Interface Code Analysis	IEEE 1012
	Well-controlled software interfaces	ANSI/ANS-10.4
16	Documentation Evaluation	IEEE 1012
17	Test Procedure Generation	IEEE 1012 & ANSI/ANS-10.4
	Unit Test	
	Integration Test	
	System Test	
	Acceptance Test	

Table A-12. Minimum V&V Tasks (cont.)

18	Unit Test Execution	IEEE 1012
	Unit test results	ANSI/ANS-10.4
19	Integration Test Execution	IEEE 1012
	Size	ANSI/ANS-10.4
	Timing	ANSI/ANS-10.4
	Interface control	ANSI/ANS-10.4
	Interactions verified	ANSI/ANS-10.4
	Build control and documentation	ANSI/ANS-10.4
20	System Test Execution	IEEE 1012
	Each requirement tested?	ANSI/ANS-10.4
	Each requirement met?	ANSI/ANS-10.4
	All test cases executed and checked?	ANSI/ANS-10.4
21	Acceptance Test Execution	IEEE 1012
22	Installation Configuration Audit	IEEE 1012
	Deliverables identified	ANSI/ANS-10.4
	Can delivered program be rebuilt?	ANSI/ANS-10.4
	Do test cases still work?	ANSI/ANS-10.4
23	V&V Final Report	IEEE 1012
24	Baseline Change Assessment (as required)	IEEE 1012
25	Review Support—participation in software and management reviews	IEEE 1012

Table A-13. Minimum Documentation Needed for *a Posteriori* V&V

1	Problem statement	ANSI/ANS-10.4
2	Requirements specification	ANSI/ANS-10.4
3	Design specification	ANSI/ANS-10.4
4	Test plan and test results	ANSI/ANS-10.4

Table A-14. Typical Policies and Directives of a Configuration Management Operation

1	Definition of software levels or classes	IEEE 828
2	Naming conventions	IEEE 828
3	Version ID conventions	IEEE 828
4	Product ID policy	IEEE 828
5	IDs of specifications, test plans, manuals & documents	IEEE 828
6	Media ID and file management	IEEE 828
7	Documentation release process	IEEE 828
8	Software release to general library	IEEE 828
9	Problem reports, change requests and orders	IEEE 828
10	Structure & operation of CCBs	IEEE 828
11	Acceptance or release of software products	IEEE 828
12	Operating rules for the software library	IEEE 828
13	Audit policy	IEEE 828
14	Methods for CCB assessment of change impact	IEEE 828
15	Level of testing or assurance required before an item is accepted for CM—may be related to software classes	IEEE 828
16	Level of SQA or V&V required before an item is accepted for CM—may be related to software classes	IEEE 828

Appendix A

Appendix B: Testing Existing Software for Safety-Related Applications

Prepared by
John A. Scott
J. Dennis Lawrence

Lawrence Livermore National Laboratory
7000 East Avenue
Livermore, CA 94550

Prepared for
U.S. Nuclear Regulatory Commission

Appendix B

ABSTRACT

The increasing use of commercial off-the-shelf (COTS) software products in digital safety-critical applications is raising concerns about the safety, reliability, and quality of these products. One of the factors involved in addressing these concerns is product testing. A tester's knowledge of the software product will vary, depending on the information available from the product vendor. In some cases, complete source listings, program structures, and other information from the software development may be available. In other cases, only the complete hardware/software package may exist, with the tester having no knowledge of the internal structure of the software.

The type of testing that can be used will depend on the information available to the tester. This report describes six different types of testing, which differ in the information used to create the tests, the results that may be obtained, and the limitations of the test types. An Annex contains background information on types of faults encountered in testing, and a Glossary of pertinent terms is also included.

Appendix B

CONTENTS

TABLES

FIGURES

Appendix B

ACKNOWLEDGMENT

The authors thank and acknowledge Professor Richard Hamlet for reviewing this report and providing helpful insights and comments.

Appendix B

TESTING EXISTING SOFTWARE FOR SAFETY-RELATED APPLICATIONS

1. INTRODUCTION

1.1. Purpose

The increasing use of commercial off-the-shelf (COTS) software products in digital safety-critical applications is raising concerns about the safety, reliability, and quality of these products. One of the factors involved in addressing these concerns is product testing. A tester's knowledge of the software product will vary, depending on the information available from the product vendor. In some cases, complete source listings, program structures, and other information from the software development may be available. In other cases, only the complete hardware/software package may exist, with the tester having no knowledge of the internal structure of the software.

The type of testing that can be used will depend on the information available to the tester. This report describes six different types of testing, which differ in the information used to create the tests, the results that may be obtained, and the limitations of the test types. An annex contains background information on types of faults encountered in testing, and a Glossary of pertinent terms is also included.

1.2. Scope, Assumptions and Limitations

This report specifically addresses testing of existing, commercial off-the-shelf software for safety-related applications and, therefore, makes no assumptions as to the adequacy of the software process under which the software was developed or of the capabilities of the software developer. These and other questions must be considered by whatever process determines the acceptability of the COTS software product for a particular use. Testing is only one aspect of an acceptance process for a COTS software product. Other aspects include a system design that carefully allocates responsibilities to the computer system, a hazard analysis of the system (including computer hardware and software), an investigation of the capabilities of the software developer, a mature development process, and favorable experience data. These aspects are discussed in the main report in this NUREG/CR, and related information is found in Lawrence (1993), Lawrence and Preckshot (1994), and Preckshot and Scott (1995). Results obtained from applying testing strategies discussed in this report will, therefore, be used in combination with data from the other information sources used in the acceptance process.

This report provides an overview of key testing techniques and their relationship to COTS software. The quoted references should be consulted for more detail. In particular, Beizer (1990) and Marick (1995) provide detailed, practical information on carrying out testing activities.

1.3. Report Organization

The body of the report consists of six sections, numbered 2–7, which describe six different testing strategies. Within each testing strategy, a number specific testing techniques are described. The testing strategies are:

- Static Source Code Analysis
- Structural Testing
- Functional Testing
- Statistical Testing
- Stress Testing
- Regression Testing.

Each of these sections is organized in a similar fashion:

- Purpose of the testing strategy
- Benefits and limitations of the testing strategy
- Information required to perform the tests
- Methods of performing the tests
- Discussion of the test techniques belonging to the testing strategy.

The sections are meant to be read independently, so some repetition of material occurs throughout sections 2–7.

An Annex has been included to provide additional information regarding the types of faults discovered

during testing, as well as a Glossary of software quality terms.

1.4. Definitions

Several terms used in this report are defined here. The Glossary provides a more complete listing of applicable terminology.

- *Commercial Off-the-Shelf (COTS) software.* COTS software is developed for general commercial use and, as such, is usually developed without knowledge of the unique requirements of particular applications. The term COTS, as used here, does not denote an acceptance process nor does it have any connotations regarding the availability of source code or development process records.
- *Operational Profile.* The operational profile of a program is the statistical distribution function of the inputs which will be encountered by the program under actual operating conditions.
- *Oracle.* Any (often automated) means of judging the correctness of a test execution.[14]
- *Software Object.* The software module, package, program, subsystem, or system which is being tested.
- *Testing*. "(1)The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component. (2) The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items." (IEEE 610.12-1990) In this report, the word "testing" is used in both meanings.

1.5. General Comments on Testing

This section contains brief comments on software testing that apply generally to the remainder of the report. Note that the tables of Section 1 should not be read as absolutes, but as general guidance. In particular cases, some connections indicated in the tables may not be relevant, and some connections that are not indicated in the tables may be important. Nevertheless, in most cases, the tables provide general guidance for testing safety-related COTS software.

[14] A more restrictive definition is given by Beizer (1990) who states, "An oracle is any program, process, or body of data that specifies the expected outcome of a set of tests as applied to a tested object. . . The most common oracle is an input/outcome oracle—an oracle that specifies the expected outcome for a specified input." This is more difficult to create and is not necessary to this report.

1.5.1. Testing Goals and Software Qualities

To be effective, testing should be directed at measuring some quality of the software. The various testing strategies address different sets of software qualities. For this reason, a comprehensive testing program will incorporate as many strategies as possible in an attempt to assess the overall soundness of the software. Within this context, special emphasis can be placed on those strategies that are related to quality attributes of particular concern.

Hetzel (1984) divides software qualities into three sets: external, internal, and future. External qualities describe the functionality of the software; internal qualities describe the engineering aspects of the software; and future qualities describe the adaptability of the software. Many possible software qualities have been described in the software engineering literature. A list of qualities collected by Hetzel (1984) and by Charette (1989) has been arranged by the likely impact of the qualities on safety in Table 1-1. Definitions of these qualities are given in the Glossary.

The six different testing strategies are not equally suited to all of the software qualities. Table 1-2 suggests which strategies to use for the qualities that are of primary and secondary interest in safety-related reactor applications. The table provides a cross reference between software qualities and strategies used to test for these qualities. These linkages can be useful to both developers and evaluators of COTS software. Regression testing attempts to ensure that changes made to the software, either during development or after installation, do not affect a software object in unplanned areas. It consists of re-execution of previous testing and, therefore, addresses the qualities previously demonstrated with other forms of testing.

1.5.2. Software Objects

Software objects subject to testing range from programming language statements to complete systems, and the type and amount of testing will generally vary across this range. To provide some consistency within this report, five classes of objects are defined. In particular instances, some classes may coalesce. For example, in the simplest case of a system consisting of a single module, all five classes are compressed into one. Most classes will be distinct in safety-critical systems.

Software object terminology is defined for conventional third-generation programming languages such as *Ada*, *C*, *C++*, *Pascal*, and FORTRAN. Extensions to fourth-generation languages and visual programming environments should be straightforward.

Table 1-1. Safety Impact of Software Qualities from a Regulator Viewpoint[15]

	Impact on Operational Safety		
	Primary Impact	**Secondary Impact**	**Little Impact**
External (Functional) Qualities	Accuracy Acceptability Availability Completeness Correctness Interface Consistency Performance (Efficiency, Timing) Preciseness Reliability Robustness Security Usability	User Friendliness	
Internal (Engineering) Qualities	Integrity Internal Consistency Testability Validity	Clarity Interoperability Simplicity Understandability	Accountability Adaptability Generality Inexpensiveness Manageability Modularity Self-Descriptiveness Structuredness Uniformity
Future Qualities			Accessibility Augmentability Convertibility Extendibility Maintainability Modifiability Portability Reparability Reusability Serviceability

[15]Note that qualities associated with modifications that might be made in the operations phase have been listed in the "Little Impact" category because an assumption is made here that, in typical safety-related reactor applications, changes will be infrequent. To the extent that such software might be used in an environment with regularly changing requirements, these qualities assume more importance. It should also be noted that, in some cases, listed qualities have essentially the same meaning but may have slightly different interpretations depending on the context. Since they all appear in the literature, no attempt has been made to group them. They are, however, categorized consistently.

Table 1-2. Testing Strategies Appropriate to Software Qualities

Software Quality	Static Analysis	Structural Testing	Functional Testing	Statistical Testing	Stress Testing
Acceptability			X		O
Accuracy	O	X	X		
Availability				X	X
Clarity	X				
Completeness	X		X		O
Correctness	X	X	X		X
Integrity	O	X	X		
Interface Consistency	X		X		
Internal Consistency	X	X	O		
Interoperability	X		X		
Performance (efficiency & timing)		O	X		X
Preciseness	O	X	X		
Reliability				X	
Robustness	O		X		X
Security	O	X	X		
Simplicity	X				
Testability	X				
Understandability	X				
Usability	X		X		
User Friendliness			X		
Validity	X		X		
Regression Testing		O	X	O	X

X = Strategy should be used for the specified quality
O = Strategy may be used for the specified quality

- A *module* is a named collection of programming language statements. Alternate names are *subroutine*, *procedure*, or *unit.*
- A *package* is a collection of one or more modules which relate to a common topic. Packages are a key feature of object-oriented programming languages such as Ada and C++. For example, a set of modules that processes dates could be combined into a calendar package. A set of modules that manages sensor information (read, check status, convert data) could be combined into a sensor device-driver package.
- A *program* is a set of one or more packages and modules which can be executed on a computer. Programs are created by means of a linker or loader and can be stored in a file or PROM[16] for future use.
- A *subsystem* consists of one or more modules, packages and programs which are devoted to one or more related functions, or which must execute together to achieve a desired task, or which execute concurrently on the same processor. Examples include a set of programs which performs various kinds of report production, and a set of programs which reads and processes sensor

[16] Programmable read-only memory.

data on one computer and sends the results to be displayed on another computer.

- A *system* is the entire set of subsystems which manages a complete application.

Table 1-3 shows a different perspective. It matches different test strategies to different classes of software objects. The checked entries show which testing strategies are primarily recommended for each class of object. Note that any strategy could apply to any class of object under specific circumstances. The table merely provides general guidance.

Objects are classified here according to structure, and this classification is used throughout the report. Another method of classification relates to structural complexity. This might yield a series such as batch processing, interactive time-sharing, transaction processing, real-time process control, and real time vehicle control. However, this report is limited to real time process control systems.

A further classification dimension involves the interaction of processes and ranges from single process systems to multiple-process shared-memory concurrent systems. This dimension affects primarily the amount of testing required and the difficulty of creating and judging the tests. In particular, stress testing is very important as the amount of interaction increases.

1.5.3. Testers

Testing is frequently carried out by different categories of personnel. A primary concern when safety is an issue is independence of testing from development.[17]

During development, the software engineer who develops code may be involved in some of the testing. Some independence can be achieved by using other software engineers from the developing organization. Greater independence can be achieved if the customer or an IV&V organization performs testing activities. In these cases, testing could be subcontracted. For example, the customer might hire a company to carry out testing on its behalf or it might do the testing itself. When COTS software is to be tested by the customer, it is unlikely that parties from the developing organization will be involved in the testing effort, so independence would generally be assured. In any case, note that it is essential that the testers be well-qualified and knowledgeable of the application.

Table 1-4 shows which categories of tester are most likely to carry out testing on the different types of software objects. As with the previous tables, exceptions do occur. For example, a programmer could carry out all testing strategies.

Table 1-5 similarly shows which categories of testers are likely to use the different testing strategies. Again, these are recommendations, not absolutes.

1.5.4. The Testing Life Cycle

Software testing has a life cycle of its own that is similar to the software development life cycle. Testing life cycle phases generally include planning, requirements, design, implementation, and operation (execution). Note that V&V activities apply to testing life cycle products (reviews of test plans & designs, etc.) in addition to software development life cycle products.

If testing is carried out by or on behalf of the development organization, the testing life cycle phases should occur concurrently with the development life cycle phases. This is not likely to be possible with customer testing of COTS software. However, the testing life cycle should still exist and be carried out.

Testing life cycle activities are described in detail in IEEE Software Engineering Standards 829 and 1074 and are not discussed here. The following list provides a brief synopsis of the activities based on these standards, assuming that the testing will be carried out by (or on behalf of) the customer.

- Test planning activities
  - Prepare test plan
- Test requirements activities
  - Determine the software qualities for which testing is required
  - Determine the software objects to be tested

[17] Independent V&V is used when it is necessary to have an impartial, objective analysis and test conducted of the software/system. The notion is that difficult-to-discover errors which may reside in the software due to assumptions or technical biases inadvertently introduced by the development team would have a higher probability of being detected by an impartial, objective V&V team who would apply a fresh viewpoint to the software. IV&V is used for high-criticality software, which demands the integrity of critical functions due to life-threatening consequences of failure, unrecoverable mission completion (e.g., space probes), safety or security compromises, financial loss, or unacceptable social consequences. Independence is defined by three parameters: technical, managerial, and financial. The degree of independence of the V&V effort is defined by the extent that each of the three independence parameters is vested in the V&V organization. The ideal IV&V contains all three independence parameters. Technical independence requires that the IV&V team (organization or group) utilize personnel who are not involved in the development of the software. An effective IV&V team has personnel who have some knowledge about the system or whose related experience and engineering background gives them the ability to quickly learn the system. In all instances, the IV&V team must formulate its own understanding of the problem and how the proposed system is solving the problem. This technical independence ("fresh viewpoint") is crucial to the IV&V team's ability to detect the subtle errors that escape detection by development testing and quality assurance reviewers. (Personal communication on work being done on the update of IEEE 1012).

– Obtain needed resources: budget, time, and assignment of personnel

Table 1-3. Test Strategies Appropriate for Software Objects

	Module	Package	Program	Subsystem	System
Source Code Analysis	X				
Structural	X			O	O
Functional	O	X	X	X	X
Statistical			X	X	X
Stress	O		X	X	X
Regression	X	X	X	X	X

X = Test strategy should be used on specified software object
O = Test strategy may be used on specified software object

Table 1-4. Expected Pattern of Testers and Software Objects

	Module	Package	Program	Subsystem	System
Software Engineer	X	X	O		
Development Organization	O	X	X	O	O
Customer				X	X
Independent Tester	O	O	X	X	X

X = Tester is likely to test specified software object
O = Tester may test specified software object

Table 1-5. Strategies Used by Testers

	Software Engineer	Development Organization	Customer	Independent Tester
Source Code Analysis	X	O		O
Structural	X	X		O
Functional	X	X	X	X
Statistical		O	X	X
Stress	O	X	X	X
Regression	X	X	X	X

X = Test strategy should be used on specified software object
O = Test strategy may be used on specified software object

- Test design activities
  - Prepare test design specifications
  - Prepare test procedures
  - Prepare test case specifications
  - Design test station
- Test implementation activities
  - Prepare test cases
  - Prepare test data
  - Create test station
- Test execution activities
  - Execute test cases
  - Analyze test results
  - Prepare test reports

1.6. Faults, Errors, Failures

One purpose of testing is to identify and correct program faults, which is done by examining program failures.

1.6.1 Definitions

A *fault* is a deviation of the behavior of a computer system from the authoritative specification of its behavior. A software fault is a mistake (also called a *bug*) in the code.

An *error* is an incorrect state of hardware, software, or data resulting from a fault. An error is, therefore, that part of the computer system state that is liable to lead to failure. Upon occurrence, a fault creates a latent error, which becomes effective when it is activated, leading to a failure. If never activated, the latent error never becomes effective and no failure occurs.

A *failur*e is the external manifestation of an error. That is, a failure is the external effect of the error, as seen by a user (human or physical device), or by another program.

1.6.2 Relationship of Faults, Errors, and Failures

Assume that the software object under test contains a fault B. Depending on the circumstances, execution of the code containing fault B may or may not cause a change of state which creates an error E. Again, depending on circumstances, E may or may not cause a failure F to occur.[18] Note that neither fault B nor error E is observable; only failure F is observable.

Dynamic testing[19] consists of presenting the software object with a sequence of inputs I and observing failures. This amounts to searching for sequences I $\rightarrow$ B $\rightarrow$ E $\rightarrow$ F. Other sequences are possible. For example:

I alone (that is, no fault is encountered),

I $\rightarrow$ B (but no error occurs),

and I $\rightarrow$ B $\rightarrow$ E (but no failure occurs).

None of these sequences can be observed from system output, although two of them do contain faults.

As an example, suppose a program contains the statement

```
x1l = (a + b) / (c + d)
```

This statement is used later on in one of two ways, depending on the value of a flag variable which is almost always true:

```
if (flag) then y = x1l - 4
          else y = x11 + 3
```

There is a fault here, since the last statement contains a typographical error - `x11` (‘ex-one-one’) is used instead of `x1l` (‘ex-one-el’). Most of the time, this does not matter, since the faulty statement is rarely executed. However, if it is executed, then variable ‘y’ will have an incorrect value, which is an error (incorrect state). As long as ‘y’ is not used, no observable harm occurs. Once ‘y’ is used later in a calculation, however, the program may perform an incorrect action, or simply fail. This action (or the program’s failure) is the failure F mentioned above.

Although the cause of the failure runs fault-error-failure, the diagnosis usually takes place in the other order: failure-error-fault. Specifically, from failure F, the activity of debugging attempts to infer the error which caused the failure; this may or may not be done correctly. The fault B must itself be inferred from the inferred error; again, this may or may not be done correctly. If the causal analyses of either of the sequences, F $\rightarrow$ E or E $\rightarrow$ B, is done incorrectly, fault B is not likely to be corrected. Worse, a correct piece of code may be inappropriately “fixed,” resulting in a new fault in the software object.

An implication of this is that any estimate of the effectiveness of a testing activity is inaccurate by an unknown (and almost certainly unknowable) amount. In particular, any estimate of the number of faults remaining in the software object which is derived from

[18] Considerable time delays may occur between these events. B could potentially cause more than one type of error, and each such error could potentially cause more than one type of failure, depending on the actual execution circumstances of the code.

[19] Static analysis, discussed in Section 2, is an attempt to discover faults directly by examining the source code.

testing is imprecise by an unknown amount. This should not be surprising—similar effects can be observed in science anytime inductive reasoning is used.

It is widely believed by software engineers that a properly designed test program can reduce the uncertainties in testing effectiveness sufficiently that they can be acceptably ignored. The operative words are "properly" and "believed." The first word is itself ill-defined, while "belief" lacks the confidence that comes with scientific or mathematical proof. A final point is that extending a general belief (that applies generally to testing) to a specific software object under test adds an additional inference of unknowable uncertainty.

These observations apply to all dynamic testing strategies discussed below except statistical testing. The latter is inherently interested in failures rather than faults, so the argument does not apply. This argument helps explain, however, why testing can never be perfect.

1.7. Selection of Testing Strategies and Techniques

This section discusses the context and goals associated with the testing of COTS software and provides guidelines for applying the various testing strategies discussed in the following sections.

1.7.1. Context for Selecting Testing Strategies

The testing of a COTS software item is normally done within the context of a larger process whose goal is to determine the acceptability or non-acceptability of the COTS software for use in a particular application. Consequently, this report does not address the issue of determining acceptance criteria for the use of a COTS software item in a particular application. It is assumed that the acceptance process will identify specific needs to be addressed with testing, that this report will serve as a reference for planning and conducting the necessary testing, and that the results will be evaluated, with other information, within the context of the acceptance process.

A COTS software item might be tested in order to gain additional information about the product itself or to examine the behavior of the product in the planned application. In general, the more important a COTS software item is to safety, the less one would expect to need after-the-fact COTS software testing to augment other information in order to demonstrate acceptability. In other words, the COTS software item should already be demonstrably well-qualified for its intended role. In this case, testing activities will probably be narrowly focused on particular qualities or attributes of the software. For items less important to safety, it may be appropriate (depending on the specifics of the acceptance process) to rely to a larger degree on after-the-fact testing, and a more comprehensive testing effort might be appropriate. Regardless of the scope of any potential testing effort, it will be useful to obtain information about past and current faults as well as configuration and operating parameters, reliability and availability, and comments about other qualities based on the experience of users of the COTS software item.

In addition to augmenting the testing effort conducted during software development, there might also be new requirements specific to the intended use of the COTS software item that should be addressed with testing. These might be related to particular safety functions to be performed, special performance constraints, adaptation to new hardware platforms, particular standards adopted for the application, or a need for demonstrating high confidence in particular software qualities. In these cases, the appropriate strategies must be selected to address the areas of concern. This testing effort could be quite extensive. For example, functional testing might be used to verify that certain functions are handled correctly, stress testing might be used to examine performance in the target environment, and statistical testing could be applied to assess reliability.

1.7.2. Considerations for Selecting Testing Strategies

This subsection provides assistance in selecting testing strategies and techniques to meet the needs defined by a COTS acceptance process. Since there may be multiple techniques that will address a particular testing question, and since it is not possible to anticipate all types of questions that might arise in various situations, the information provided must be considered as guidance rather than as a prescriptive formula. It should also be noted that this section refers to traditional third-generation languages (e.g., Ada, C, C++, Fortran, and Pascal) and does not necessarily apply specialized or developing technologies such as artificial intelligence systems.

The process of selecting testing strategies for a COTS software item is constrained by the information available. Table 1-6 presents a summary of the minimum information required for the various testing strategies. Representative information is also provided regarding the extent of testing to be applied when using a particular testing strategy; refer to the appropriate section for more detail. Table 1-6 provides a first-order estimate of the prerequisites and scope of a testing effort. Each situation is unique and the reviewer should refer to the text and other references to make determinations regarding the nature and extent a specific testing effort. The terminology used in Table 1-6 is explained in later sections of this report.

Table 1-7 presents a set of questions about software qualities that can be addressed by selected testing strategies. The table is not exhaustive. However, it provides useful examples for selecting testing strategies to meet specific testing requirements. The taxonomy of faults presented in the Annex is also helpful in selecting testing strategies. The terminology used in Table 1-7 is explained in later sections of this report.

Table 1-6. Sample Prerequisites for and Extent of Testing

Strategy: Technique	Goal	Minimum Information Required	Suggested Extent of Testing/Analysis
Static:			
Inspection (I0)	Examine architectural design with requirements as reference	Software requirements; architectural design	One or more inspections. Group decision on re-inspection based on inspection results.
Inspection (I1)	Examine detailed design with architectural design as reference	Architectural & detailed design	One or more inspections. Group decision on re-inspection based on inspection results.
Inspection (I2)	Examine source code with detailed design as reference	Source code & detailed design	One or more inspections. Group decision on re-inspection based on inspection results.
Inspection (other)	Check code for specific qualities, properties, or standards adherence (can be part of I2)	Source code	One or more inspections. Group decision on re-inspection based on inspection results.
Inspection (other)	Verify allocation of software requirements	System requirements & software requirements	One or more inspections. Group decision on re-inspection based on inspection results.
Inspection (other)	Check application-specific safety requirements	System & software safety requirements; hazard/risk analyses	One or more inspections. Group decision on re-inspection based on inspection results.
Desk checking	Verify key algorithms & constructs	Source code	One pass per revision; continue until no new faults are found.
Automated structural analysis	Produce general/descriptive information; compute metrics values	Source code	One pass per revision
Automated structural analysis	Fault detection	Source code	One pass per revision; continue until no new faults are found.
Automated structural analysis	Standards violations	Source code	One pass per revision; continue until no new faults are found.

Table 1-6. Sample Prerequisites for and Extent of Testing (cont.)

Strategy: Technique	Goal	Minimum Information Required	Suggested Extent of Testing/Analysis
Structural:			
Path	Verify internal control flow	Source code; module design specification	Branch coverage
Loop	Verify internal loop controls	Source code; module design specification	Focus on loop boundaries
Data flow	Verify data usage	Source code; module design specification	All-'definition-usage'-pairs
Domain (structural)	Verify internal controls/computations over input domains	Source code; module design specification	Focus on boundaries
Logic (structural)	Verify internal logic (implementation mechanisms)	Source code; module design specification	All combinations of conditions
Functional:			
Transaction	Verify implementation of application functions	Executable, software requirements	All transactions
Domain	Verify functional controls/computations over input domains	Executable, software requirements	Representative domain values including boundary and illegal values
Syntax	Verify user interface and message/signal constructs	Executable, software requirements	All input/message constructs
Logic	Verify implementation of the logic of the real-world application	Executable, software requirements	All combinations of real-world conditions
State	Verify implementation of states associated with the real-world application	Executable, software requirements	All states/transitions
Statistical	Estimate reliability	Executable, software requirements, operational profiles	Predetermined reliability target
Stress	Examine robustness; characterize degradation with increasing loads on resources	Executable, software requirements	One pass per resource per revision per operating mode; sampling of combinations of resource loads
Stress	Find breaking points; check recovery mechanisms	Executable, software requirements	Continue testing a resource until failure & recovery modes are well understood
Regression	Verify that changes have not impacted the software in unexpected ways	Various input needed depending on test strategies used in the regression test suite	Continue until no new failures are detected

Table 1-7. Typical Testing Strategies for Investigating Software Qualities

Software Quality	Also see:	Question to be Answered	Applicable Testing Strategies
Acceptability	Validity	Are real-world events handled properly?	Functional (T,D,L,Se)
		How does the product perform in realistic, heavy load situations?	Stress
Accuracy	Preciseness	Are internal calculations accurate?	Structural (DF)
		Are results accurate?	Functional (T)
		Is there confidence that important calculations are accurate?	Static analysis (I,DC)
Availability	Reliability	Will the software be unavailable due to poor reliability?	Statistical
		Will functions be available during heavy load situations?	Stress
Clarity	Understand-ability	Is the implementation sufficiently clear to a knowledgeable reviewer?	Static analysis (I,DC)
Completeness		Are all requirements expressed in the design?	Static analysis (I)
		Are all design elements implemented in the code?	Static analysis (I)
		Are internals complete? (no missing logic, undefined variables, etc.)	Static analysis (ASA,I)
		Are all aspects of real-world transactions implemented?	Functional (T)
		Are boundary values and all combinations of conditions accounted for?	Functional (D,L,Se)
		Are recovery mechanisms implemented?	Stress
Correctness		Does the product have statically detectable faults?	Static analysis (All)
		Is the implementation/modification structurally correct?	Structural (All)
		Is the implementation/modification functionally correct?	Functional (All)
		Does the product perform correctly in heavy load situations?	Stress
		Have modifications had unintended effects on the behavior of the software?	Regression
Integrity	Security	Are access control schemes appropriate?	Static analysis (I)
		Are access controls and internal protections correctly implemented?	Structural (All)
		Is end-user access management correct?	Functional (T)
		Are access-related boundary values, logic, states, & syntax correctly implemented?	Functional (D,Sx,L,Se)

Legend:

ASA	Automated Structural Analysis	I	Inspection	Se	State Testing
D	Domain Testing	L	Logic Testing	Sx	Syntax Testing
DC	Desk Checking	Lp	Loop Testing	T	Transaction Testing
DF	Data Flow Testing	P	Path Testing		

Table 1-7. Typical Testing Strategies for Investigating Software Qualities (cont.)

Software Quality	Also see:	Question to be Answered	Applicable Testing Strategies
Interface Consistency	Internal Consistency	Have interface standards & style been followed?	Static analysis (ASA,I)
		Is parameter & variable usage consistent across interfaces?	Static analysis (ASA,I)
		Is transaction data handled consistently among modules?	Functional (T)
		Are boundary conditions treated consistently?	Functional (D)
		Is message syntax consistent?	Functional (Sx)
		Is decision logic consistent among modules?	Functional (L)
		Are system states consistently treated among modules?	Functional (Se)
Internal Consistency	Interface Consistency	Have standards & style been followed?	Static analysis (ASA,I)
		Is parameter & variable usage consistent?	Static analysis (ASA,I)
		Are conditions handled consistently with respect to control flows?	Structural (P, Lp,D,L)
		Are there inconsistencies in data handling? (typing, mixed mode, I/O compatibilities, etc.)	Structural (DF)
		Are real-world events and logic handled consistently?	Functional (L, Se)
Inter-operability		Does the architecture facilitate interoperability?	Static analysis (I)
		Do modules used in transactions exchange & use information properly?	Functional (T,D,Se)
Performance		Is intra-module timing within specification?	Structural (P,Lp)
		Are transactions performed within required times?	Functional (T)
		Are timing requirements met when boundary values are input?	Functional (D)
		Is system performance adequate under heavy load conditions?	Stress
Preciseness	Accuracy	Will internal representations yield required precision?	Static analysis (DC)
		Are internal calculations sufficiently exact?	Structural (DF)
		Are real-world transaction results sufficiently exact?	Functional (T)
Reliability	Availability	What is the probability of running without failure for a given amount of time?	Statistical

Legend:

ASA	Automated Structural Analysis	I	Inspection	Se	State Testing
D	Domain Testing	L	Logic Testing	Sx	Syntax Testing
DC	Desk Checking	Lp	Loop Testing	T	Transaction Testing
DF	Data Flow Testing	P	Path Testing		

Table 1-7. Typical Testing Strategies for Investigating Software Qualities (cont.)

Software Quality	Also see:	Question to be Answered	Applicable Testing Strategies
Robustness		Has appropriate recovery logic been implemented?	Static analysis (I)
		Are poorly specified/invalid transactions handled correctly?	Functional (T,Sx)
		Are marginal/illegal inputs handled correctly?	Functional (D,Sx)
		Are unexpected combinations of conditions/states handled correctly?	Functional (L,Se)
		Can the system continue operating outside of normal operating parameters?	Stress
Security	Integrity	Are access controls properly designed/implemented?	Static analysis (I)
		Are access controls consistent with the operating environment?	Static analysis (I)
		Are the structural aspects of access control mechanisms correct?	Structural (All)
Security (continued)		Do access management functions work correctly?	Functional (T)
		Do access management functions work correctly in the presence of marginal or illegal values and constructs?	Functional (D,Sx,L,Se)
Simplicity		Are implementation solutions overly complex?	Static analysis (I)
		Are complexity-related metric values reasonable for a given situation?	Static analysis (ASA,I)
Testability		How can aspects of the software be tested?	Static analysis (DC,I)
Understand-ability	Clarity	Is the designer/implementer intent clear?	Static analysis (DC,I)
		Does information characterizing the software make sense?	Static analysis (ASA,I)
Usability	User friendliness	Can the user correctly form, conduct, & interpret results of transactions?	Functional (T,D,Sx)
		Does the user interface design support operational procedures?	Static analysis (I)
User friendliness	Usability	Is the user comfortable in forming, conducting, and interpreting results of transactions?	Functional (T,Sx)
Validity	Acceptability	Are requirements traceable?	Static analysis (I)
		Are implementation solutions appropriate?	Static analysis (DC,I)
		Is the real world appropriately represented?	Functional (All)
		Is the implementation/modification structurally correct?	Structural (All)
		Is the implementation/modification functionally correct?	Functional (All)

Legend:

ASA	Automated Structural Analysis	I	Inspection	Se	State Testing
D	Domain Testing	L	Logic Testing	Sx	Syntax Testing
DC	Desk Checking	Lp	Loop Testing	T	Transaction Testing
DF	Data Flow Testing	P	Path Testing		

Appendix B

2. STATIC SOURCE CODE ANALYSIS

2.1. Purpose of Static Source Code Analysis

Static source code analysis is the examination of code by means other than execution, either manual or automated, with the intent of (1) producing general, metric-related, or statistical information about a software object, (2) detecting specific types of faults in a software object, (3) detecting violations of standards, or (4) verifying the correctness of a software object. Static analysis pertains to certain categories of faults and should be considered complementary to dynamic testing in the overall testing effort. The qualities addressed by static analysis, summarized in Table 1-2, are discussed below.

The section is primarily focused on static source code analysis; however, some techniques, such as inspection, have broader applicability. Some of these extensions are discussed below.

2.2. Benefits and Limitations of Static Source Code Analysis

Static analysis is code examination without code execution. This approach provides a different way of thinking about fault detection and, therefore, static analysis techniques are best applied as part of an overall testing (or verification and validation) program that also includes extensive dynamic testing. The advent of automated, interactive software environments and testing tools is blurring the distinction between dynamic testing and static analysis somewhat. In some of these tools, results are available from static examinations carried out in support of dynamic, structural testing. The use of interpreters as code is being examined can automate the desk checking technique of stepping through lines of code and, therefore, can produce information about run-time states (although this information may also be related to the use of the interpreter).

2.2.1. Benefits

There are a number of features of static analysis techniques that make them an effective complement to dynamic techniques. The inspection or review-oriented techniques have the advantage of combining the different perspectives of the participants and can produce fault information that may be overlooked by a single examiner. Inspections have been found to be very effective in detecting the types of faults that can be found with static techniques. In addition, manual static analysis techniques can easily incorporate project-specific standards adopted for the application of a COTS item to a particular use. Automated structural analyzers can perform large numbers of static checks that could not be performed manually, and may detect structural faults that might go undetected in dynamic testing since all possible paths cannot be covered by test cases. Static analysis techniques that provide general information about software objects can produce information that will be valuable in developing test cases for dynamic testing.

Regarding the assessment of software qualities, static analysis techniques are effective in examining software for possible faults related to completeness, consistency, and validity. For example, information about the completeness of a software item can be gained from automated structural analyses that discover missing logic, unreachable logic, or unused variables. Inspections can provide information about the traceability of requirements. Both inspections and automated structural analyses provide a means for evaluating the consistency of application of standards and style guidelines, as well as for checking parameter and variable usage from a static perspective. Desk checking can provide information about the accuracy and precision of algorithm implementations.

Static analyses also provide information about other software qualities that may be important to the intended use of a COTS software item, including testability, usability, interoperability, clarity, understandability, robustness, and simplicity. These tend to be areas where judgment is required, making the manual techniques particularly effective. The qualities are addressed with the manual techniques by including the appropriate considerations in inspection checklists or desk-checking tasks. For example, the number of questions about intent raised during an inspection is an indicator of understandability. In addition, automated structural analyses can provide metrics and structural information that is useful in assessing these software qualities.

2.2.2. Limitations

Static analysis techniques, in general, do not provide much information about run-time conditions. In addition, many of these techniques are labor-intensive and, therefore, can be quite expensive to carry out. In cases where there are project-specific considerations that need examination by an automated tool, automated analyzers must be developed, which is also a costly endeavor.

2.3. Information Required to Perform Static Source Code Analysis

As a minimum, the source code must be available. For most static analysis techniques to be effective, it is also necessary to have information on the context (intended usage), requirements, and design of the software object being examined. To select effective approaches for static analysis, it is useful to know what static analysis capabilities were applied in the development environment. In particular, most compilers perform various types of automatic static checking. In many cases, this checking is limited to those checks that support the compiler's primary goal of detecting syntactic faults before translating statements to object code. Compiler results such as syntactic correctness, uninitialized variables, cross reference listings and similar matters are a very useful part of static analysis, but should be considered as the first step in static analysis, not the totality. Information on the compiler checks performed is useful in determining the relative emphasis to place on the various other techniques that might be applied.

Since one goal of static source code analysis is to detect violations of standards, it is necessary to have information regarding the standards applied during the development effort. This information may be difficult to obtain for COTS software; however, some information, such as language standards or the compiler used, should be available. Perhaps a more important application of standards checking is the development (by the testing or customer organization) of required standards regarding what is acceptable for the particular COTS software application. For example, if certain language constructs are permitted by the language standard but are known to be troublesome in past practice, a safety-critical application might require a local practice standard that prohibits their use.

2.4. Methods of Performing Static Source Code Analysis

The static analysis of source code for a software object must be planned, designed, created, executed, evaluated, and documented.

2.4.1. Static Analysis Planning and Requirements

The following actions are required to plan and generate requirements for static analysis of software objects.

1. Determine the software qualities to be evaluated with static techniques. Qualities typically examined in static source code analysis are shown in Table 1-2. For the static analysis of safety-related COTS software, the primary quality of interest is correctness, particularly as it is related to the qualities of completeness, consistency, and validity. Other qualities, that may be of interest, depending on the intended role of the COTS software item, can be assessed with static analysis. These include testability, usability, interoperability, clarity, understandability, and simplicity.
2. Determine which static analysis techniques will be required. Code inspections and automated structural analyzers are recommended as a minimum.
3. Determine what resources will be required in order to carry out the analyses. Resources include budget, schedule, personnel, equipment, analysis tools, and the platform for automated structural analyses.
4. Determine the criteria to be used to decide how much static analysis will be required. This is a stopping criterion—how much analysis is enough?
5. Determine the software objects to be examined.

2.4.2. Analysis Design and Implementation

The following actions are required to design and implement static analyses.

1. Create procedures for carrying out the analyses. For techniques such as code inspection, this involves tailoring the technique to the particular project environment. For other static source code analyses, the procedures will specify analyses to be applied.
2. Prepare for the orderly and controlled application of the individual analyses. The following information should be prepared for each analysis:
    a. *Analysis identification*. Each analysis must have a unique identifier.
    b. *Purpose*. Each analysis must have a specific reason for existing. Examples include the application of an automated standards auditor to a block of code or the examination of a block of code to determine whether a particular error-prone construct has been used.
    c. *Input data*. The precise data, if any, required in order to initiate the analysis must be specified. This should include any parameter values needed by automated analyzers (this information may also be appropriate as part of the procedures).
    d. *Initial state*. In order to reproduce an analysis, the initial state of the automated analyzer may need to be specified.

e. *Results*. The expected results of the analysis must be known and specified. This could include the absence of a detection of the fault being targeted or the specific value range of a metric.

3. Create the platform to support the automated structural analyses. This is a mechanism for selecting, executing, evaluating, and recording the results of analyses carried out by automated static analyzers on the software object.[20] An automated structural analyzer might perform a pre-programmed set of checks or might require input to select specific checks (as with an interactive tool). Platform components, illustrated in Figure 2-1, include:

   a. *Analysis case selection*. A means of selecting analysis cases (checks) to be executed is required. This information may be kept in a file or database, and the selection may simply consist of "get next analysis case."

   b. *Analyzer program*. A means of setting the analyzer's initial state (if necessary), providing input to the analyzer, and recording the output from the analyzer is required.

   c. *Results database*. A means of recording the results for future analysis and evaluation is needed. Typical data to be captured include the analysis identifier, date, version of module, analysis output, and an indication of the acceptability of the results.

2.4.3. Execution and Evaluation of the Analyses

The procedures must be carried out and analyzed. If a fault is indicated in the software object and the development organization is performing the analysis, the software engineer is expected to correct the fault. The pattern of test–fix–test–fix continues until all discrepancies have been resolved.

In the case of COTS, obtaining corrections may be very difficult. Suppose the analysis is being performed by (or on behalf of) the customer. If the software was developed for the customer under contract, there should be considerable leverage for obtaining corrections. If the software is a consumer product (for example, a library accompanying a compiler used for development), experience shows that many developers have little interest in expensive repairs that satisfy a limited marketplace. In this case, the options of the customer may be simply to reject the software or to evaluate each fault detected and determine its effects on safety. If more than one fault exists, the cumulative effect of all the faults on safety must also be determined.

The nature of the faults encountered must also be considered. The discovered faults might be related to new requirements or standards arising from the specific, intended application of the COTS product. They might also be minor faults that might have escaped detection during product development. In these cases, the significance of the faults should be evaluated and the options for obtaining corrections might be pursued. However, if one or more serious faults pertaining to the product itself are discovered, confidence decreases rapidly regarding the suitability of the product for use in a safety-related application.

2.5. Discussion of Static Source Code Analysis

Static source code analyses, whether done totally manually or supported by automated techniques, are typically manpower-intensive processes. Manual processes such as inspections require team efforts. Many of the computer-aided methodologies require the involvement of the development team, the development of project-specific tools, or on-line use of interactive tools.

It should be noted that, although these techniques involve high manpower costs, static analysis techniques are effective in detecting faults. One controlled experiment (Basili and Selby, 1987) found that code reading detected more software faults and had a higher fault detection rate than did functional or structural testing. Since static analysis and dynamic testing detect different classes of faults, a comprehensive effort should employ as many static and dynamic techniques as are practical for the specific project. The remainder of this section discusses various static analysis techniques.

2.5.1. Inspection

Among the manual techniques, code inspection, peer reviews, and walkthroughs are effective methods for statically examining code. The techniques are essentially similar in that teams of programmers perform in-depth examinations of the source code; however, code inspections are distinguished by the use of checklists and highly structured teams. One of the important benefits common to these techniques is that the different perspectives and backgrounds of the participants help uncover problems that the original software engineer overlooked. All three techniques benefit from the participation of development team members and probably lose some effectiveness if these

[20] The process of selecting and initiating analyses and evaluating the results might be a manual activity; in this case the platform described is largely conceptual, although the databases should exist and be controlled.

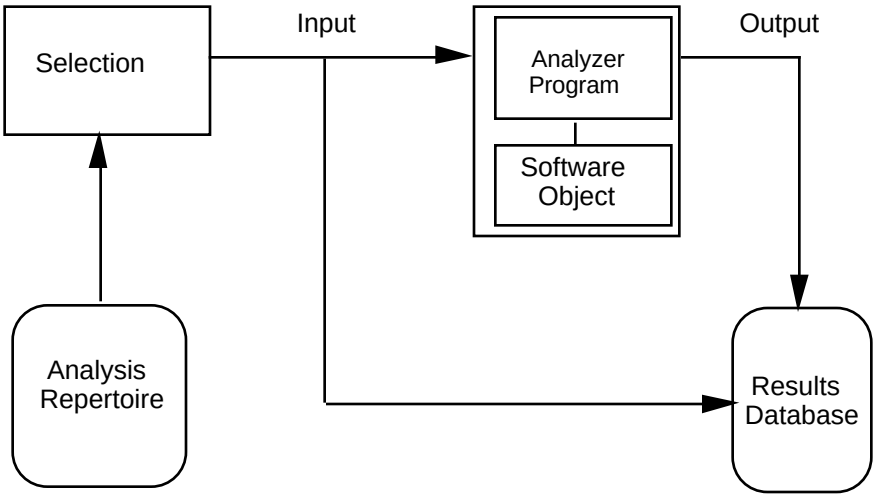


Figure 2-1. Conceptual Platform for Automated Static Analysis

members are not present, which is likely to be the case with COTS software. However, careful attention to the development and tailoring of checklists for a particular COTS application, along with the high degree of structure provided by the inspection process, should make source code inspections a valuable static analysis technique for COTS software. Peer reviews and walkthroughs are not discussed further here; information on how to perform structured walkthroughs can be found in Yourdon (1989).

Fagan (1976) provides the definitive work on inspections, a technique that can apply to a wide range of products. Inspections are defined for three points, labeled I0, I1, and I2, in the programming process. Fagan inspections that inspect against the software requirements are called I0 inspections. These inspections would typically be performed as part of the software design activities, as described in NUREG/CR-6101 (Lawrence, 1993). I1 inspections are typically performed as part of the software design activities and inspect against high-level software architectural design. I2 inspections are performed during software implementation and inspect implemented code. Figure 2-2 shows the relationship between software activity, product, and inspection type.

I0 inspections typically examine the set of unit and program designs, and their interactions to determine whether the functional content is consistent with the specified software requirements. Of particular interest for this inspection are data flows among system components and potential processing deadlocks. I1 inspections target design structure, logic, and data representation based on the previously inspected high-level design. I2 inspections focus on the translation of the detailed design into code and compliance with standards, and are commonly referred to as source code inspections. Depending on the information available about a COTS software product, any of the inspections described can be an effective technique for examining the product. In evaluating a COTS software product for use in a safety-related application, the inspection technique is useful in examining the allocation of system requirements to software and in comparing these software requirements to the capabilities of the COTS product.

All inspections follow a specific process containing planning, overview, pre-inspection preparation, inspection, rework, and follow-up phases. The follow-up phase might consist of a complete re-inspection if significant rework is required. Specific roles must be defined for an inspection; a typical team might include the designer, coder, tester, and a trained moderator. Additional perspectives of value are those of a code maintainer, user, standards representative, and application expert. The actual inspections require intense concentration and, therefore, are usually performed on small amounts of material during short (1- to 2-hour) inspection sessions. Published experience (Dyer 1992) indicates that 50 to 70 percent of faults can be removed by the inspection process (i.e., employing I0, I1, and I2 inspections).

Most discussions of source code inspections focus on the use of the technique during the development process. For COTS software, a source code inspection would be performed well after development and would involve teams of programmers not involved in the original development. Therefore, particular attention should be given to the tasks of developing an effective checklist and establishing a set of standards specific to the particular application of the COTS software. Any standards and checklists that were applied during development are a good starting point. Myers (1979)

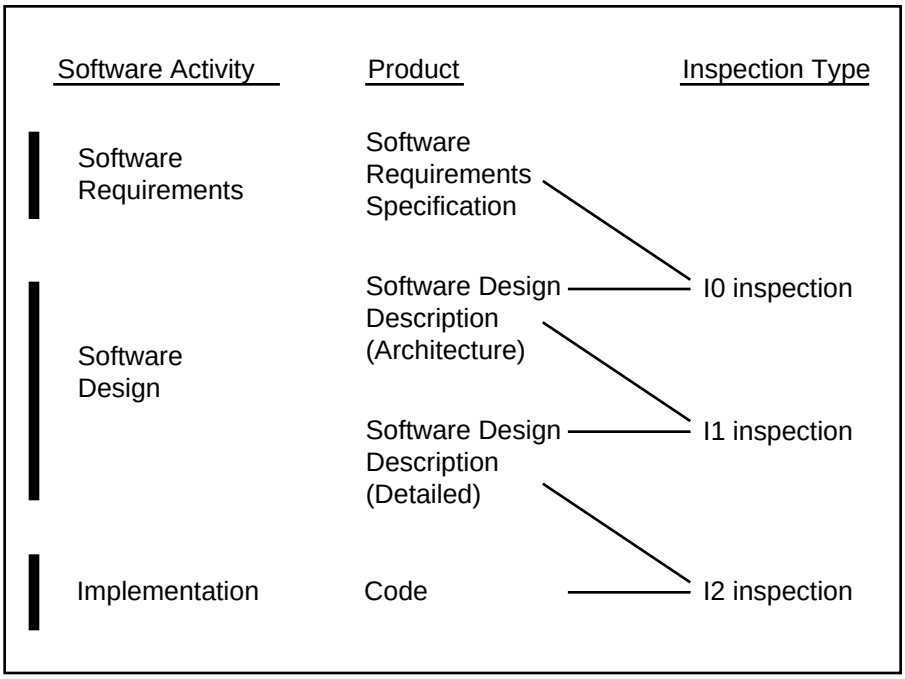


Figure 2-2. Software Development Activities, Products, and Inspections

gives a set of typical checklist items grouped by data reference faults, data declaration faults, computation faults, comparison faults, control flow faults, interface faults, and input/output faults. This serves as a starting point; the list should then be enhanced by specific knowledge about the product and application in question.

The purposes of performing after-the-development source code inspections on COTS software are to detect previously undetected faults, to ensure that dangerous practices have not been used, to discover whether undocumented features are present, and to focus on anything special pertaining to the use of the COTS application in a specific environment. In planning for static analysis, strategies should be developed for applying techniques efficiently given project resources and constraints (subject to the requirements of the commercial dedication process). The entire COTS item should be inspected if possible. If not, the focus should be directed toward key functional areas with some additional random inspections. A powerful practice with any testing or evaluation technique is to attempt to classify detected faults or observed failures (such as might have been seen in other uses of the COTS item) and then to re-examine the code, searching specifically for other instances of the fault class.

Establishing standards for a source code inspection of a COTS item is particularly important. Depending on the criticality of the particular use of the COTS item, it may be useful to start with a typical set of standards for the computer language in question and then to augment this set with additional standards based on what is known about the application in which the COTS item will be used. For example, a code unit might have been produced according to an established language standard. It might also be known that certain legitimate constructs are prone to errors. For the purposes of the COTS inspection, taking into account the intended use of the item, a requirement preventing the use of the construct might be added to the set of coding standards. In this case, the particular COTS item might be found unsuitable for the particular intended use. As an alternative, the discovery of the usage of the construct might trigger separate static analyses or dynamic tests focused on that area.

2.5.2. Desk Checking

Desk checking is a proven, primarily manual, static analysis technique. It typically involves one programmer examining code listings for faults (code reading), checking computations by independent means, and stepping through lines of code. To the extent possible, desk checking should not consist of manually performed activities that could be automated. For example, an automated standards checker could be run and desk checking could be used to confirm or justify violations. Desk checking tends to concentrate on special problems or considerations posed by the application and involves techniques appropriate to those problems or considerations. This process can be

aided with the use of interactive debuggers, interactive analysis tools, or interactive analysis features of software development environments. Regardless of which tools are used to aid the process, strategy and procedures must be developed for the systematic evaluation of the code. In addition to the discovery of specific faults, the results obtained in desk checking should also be used to help tailor the standards and checklists used in future source code inspections.

2.5.3. Automated Structural Analysis

Automated structural analysis is the use of an automated checker to examine source code for faults occurring in data and logic structures. An automated structural analyzer can be focused to detect specific faults in the code, or can produce general information about the code, such as cross-reference maps of identifiers, calling sequences, and various software quality metrics. Information in the general category is useful as reference data in the inspection and desk checking analyses discussed above. An automated structural analyzer looks for faults such as those listed below (Glass 1992):

- Undeclared or improperly declared variables (e.g., variable typing discrepancies)
- Reference anomalies (e.g., uninitialized or initialized but unused variables)
- Violations of standards (language and project standards)
- Complex or error-prone constructs
- Expression faults (e.g., division by zero)
- Argument checking on module invocations (number of arguments, mismatched types, uninitialized inputs, etc.)
- Inconsistent handling of global data
- Unreachable or missing logic.

Automated structural analyzers are typically language-specific and possibly project-specific. Discussions of some of the techniques used by structural analyzers are contained in Section 3.4 of this report. Price (1992) provides information on static analysis tools. Typical automated tools include:

- Code auditors (standards and portability)
- Control structure analyzers (calling graphs, branch and path analysis)
- Cross-reference generators
- Data flow analyzers (variable usage)
- Interface checkers
- Syntax and semantic analyzers
- Complexity measurement analyzers.

An approach for performing automated structural analysis on COTS software would be as follows:

- Determine which software qualities are to be investigated.
- Determine, if possible, what static analysis capabilities were applied in the development of the code (e.g., compiler checks).
- Determine what COTS structural analysis tools are available for the language used (and particular language standard if more than one exists) by the target COTS software.
- Select and apply the appropriate language-specific tools.
- Determine whether there are project-specific considerations that should be checked using an automated structural analyzer.
- Develop and apply the project-specific analyzer (it may be possible to structure the use of the capabilities of an interactive analysis tool to get at these issues).

2.5.4. Other Methods

Various other methods for static source code analysis have been researched. Some are mentioned briefly here but are not felt to be practical for the static analysis of COTS software at this time, either because the methods are integrated into the development process or because extensive development work would be required to implement the method.

Proof of correctness is a process of applying theorem-proving concepts to a code unit to demonstrate consistency with its specification. The code is broken into segments, assertions are made about the inputs and outputs for each segment, and it is demonstrated that, if the input assertions are true, the code will cause the output assertions to be true. Glass (1992) states that the methodology is not yet developed enough to be of practical use, estimating that practical value for significant programs is about 10 years away. Advantages, if the method is practical, include the use of a formal process, documentation of dependencies, and documentation of state assumptions made during design and coding. Proof of correctness is a complex process that could require more effort than the development itself.

Symbolic evaluation is a technique that allows variables to take on symbolic values as well as numeric

values (Howden 1981). Code is symbolically executed through a program execution system that supports symbolic evaluation of expressions. Passing symbolic information through statements and operating symbolically on the information provides insights into what a unit is actually doing. This technique requires a program execution system that includes symbolic evaluation of expressions and path selection. One application of this technique would be an attempt to determine if a formula or algorithm was correctly implemented.

Automated structural analyzers are usually based on pre-defined sequences of operations. An extension to automated structure analyzer capabilities would be to develop mechanisms whereby user-specifiable sequences could be defined for subsequent analysis. Olender (1990) discusses work to define a sequencing constraint language for automatic static analysis and predicts its value when embedded in a flexible, adaptable software environment.

Appendix B

3. STRUCTURAL TESTING

3.1. Purpose of Structural Testing

Structural testing (also known as “white box” or “glass box” testing) is conducted to evaluate the internal structure of a software object. The primary concerns of structural testing are control flow, data flow, and the detailed correctness of individual calculations. Structural testing is traditionally applied only to modules, although extensions to subsystems and systems are conceivable. It is generally carried out by the software engineer who created the module, or by some other person within the development organization. For COTS software, personnel from the development organization will probably not be available; however, structural testing can be carried out by an independent test group. The qualities addressed by structural testing, summarized in Table 1-2, are discussed below.

3.2. Benefits and Limitations of Structural Testing

Both the benefits and the limitations of structural testing are effects of the concentration on internal module structure. Structural testing is the only method capable of ensuring that all branches and loops in the module have been tested. There are important classes of faults that are unlikely to be discovered if structural testing is omitted, so no combination of the other test methods can replace structural testing.

3.2.1. Benefits

Beizer (1990) states that path testing can detect about one-third of the faults in a module. Many of the faults detected by path testing are unlikely to be detected by other methods. Thus path testing is a necessary but not sufficient component of structural testing. A combination of path and loop testing can uncover 50 to 60% of the intra-modular faults. Adding data flow testing results, on average, in finding nearly 90% of intra-module faults. (It is assumed here that a thorough testing effort is performed with respect to each technique.) Some modules, of course, are worse than average, and the remaining faults are likely to be particularly subtle.

Structural testing is focused on examining the correctness of the internals of a module, i.e., on faults relating to the manner in which the module was implemented. This includes faults related to accuracy, precision, and internal consistency. Control flow faults based on inconsistent handling of conditions can be found, as well as data inconsistencies related to typing, file I/O, and construction of expressions. Some information, such as algorithm timing, can be gained regarding software performance. Finally, emphasis on testing proper referencing and data handling as well as on the implementation of access controls provides information about integrity and security.

3.2.2. Limitations

Structural testing is impossible if the source code is not available. The modules must be well understood for test cases to be designed and for correct results of the test cases to be predictable. Even moderately large collections of well-designed modules benefit from the assistance of reverse engineering tools, test generators, and test coverage analysis tools. Generating an adequate set of structural test cases is likely to be quite time-consuming and expensive.

Structural testing is almost always restricted to testing modules. Given further research, it might be possible to extend structural testing to subsystems and systems, which would be useful for a distributed control system (DCS). Here, the analogy to the flow of control among the statements of a module is the flow of control that takes place as messages are passed among the processes making up the DCS. When concurrent communicating processes are executing on a network of different computers, subtle errors involving timing can occur, and structural testing might be extended to help detect these.

3.3. Information Required to Perform Structural Testing

Structural testing requires detailed knowledge of the purpose and internal structure of the module: module specification (including inputs, outputs and function), module design, and the source code.

A test station is recommended. This station would have the ability to select pre-defined test cases, apply the test cases to the module, and evaluate the results of the test against pre-defined criteria.

3.4. Methods of Performing Structural Testing

The structural test must be planned, designed, created, executed, evaluated, and documented.

3.4.1. Test Planning and Test Requirements

The following actions are required to plan and generate requirements for structural testing.

1. Determine the software qualities that are being evaluated. For the structural testing of safety-related COTS software, the primary quality of interest is correctness, particularly in the sense of accuracy, precision, robustness, and internal consistency.

2. Determine which structural testing techniques will be required. Control flow (path) and data flow testing are the minimum requirements. Additional techniques may be required in some cases.

3. Determine what resources will be required in order to carry out the testing. Resources include budget, schedule, personnel, equipment, test tools, test station, and test data.

4. Determine the criteria to be used to decide how much testing will be required. This is a stopping criterion—how much testing is enough? For example, "95% of all paths in the module shall be covered by control flow testing."

5. Determine which modules will be tested.

3.4.2. Test Design and Test Implementation

The following actions are required to design and implement structural testing.

1. Create procedures for executing the structural test cases. This is typically done within the context created by test plan and test design documents (IEEE 829). Additional guidance for the testing process for modules is given in IEEE 1008.

2. Create individual test cases. Each test case should contain the following information:
   a. *Test identification.* Each test case must have a unique identifier.

   b. *Purpose.* Each test case should have a specific reason for existing. Examples include executing a specific path through the module, manipulating a specific data object, or checking for a specific type of fault. For the latter, see headings 3 and 4 of the Bug Taxonomy in the Annex.

   c. *Input data.* The precise data required in order to initiate the test must be specified.

   d. *Initial state.* In order to reproduce a test case, the initial state of the module (before the test begins) may need to be specified. This information is not necessary if the module is intended to execute correctly and identically in all initial states. For example, a square root module should return the square root of the input value no matter what has gone before.

   e. *Test results.* The expected results of the test must be known and specified. These can include values of data objects external to the module (such as actuator values and database values) and values of output parameters returned through the module calling interface.

   f. *Final state.* In some cases, the final state of the module must be specified as part of the test case information. This can occur, for example, if the final state after a call is used to modify the execution of the module the next time it is called.

3. Create the test station. This is a mechanism for selecting, executing, evaluating, and recording the results of tests carried out on the module. Test station components, illustrated in Figure 3-1, include:
   a. *Test case selection.* A means of selecting test cases to be executed. Test case information is typically kept in a file or database, and the selection may simply consist of "get next test case."

   b. *Test program.* A means of setting the module's initial state (if necessary), providing input to the module, recording the output from the module and (if necessary) recording the final state of the module.

   c. *Test oracle.* A means of determining the correctness of the actual output and module state.

   d. *Results database.* A means of recording the test results for future analysis and evaluation. Typical data are: test identifier, date, version of module being tested, test output and state, and an indication of correctness or failure of the test.

3.4.3. Test Execution and Test Evaluation

The test procedures must be carried out and the results analyzed. If discrepancies between the actual and expected results occur, there are two possibilities: either the test case has a fault or the module has a fault. In the first case, the test case should be corrected and the entire test procedure rerun.

If the module has a fault and the development organization is performing the test, the programmer is expected to correct the fault. The pattern of test–fix–test–fix continues until all discrepancies have been resolved.

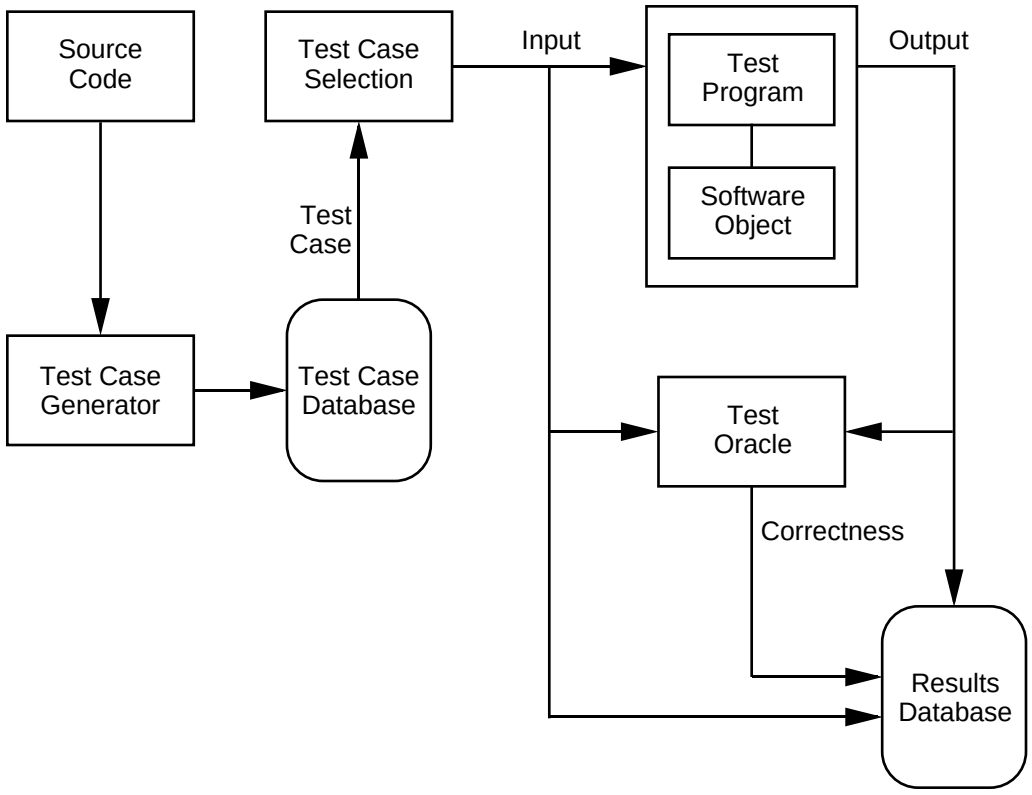


Figure 3-1. Typical Test Station Components for Structural Testing

In the case of COTS, obtaining corrections may be very difficult. Suppose the test is being performed by (or on behalf of) the customer. If the software was developed for the customer under contract, there should be considerable leverage for obtaining corrections. If the software is a consumer product (for example, a library accompanying a compiler used for development), experience shows that many developers have little interest in expensive repairs that satisfy a limited marketplace. In this case, the options of the customer may be simply to reject the software or to evaluate each fault detected by the testing and determine its effects on safety. If more than one fault exists, the cumulative effect of all the faults on safety must also be determined.

The nature of the faults encountered must also be considered. The discovered faults might be related to new requirements arising from the specific, intended application of the COTS product. They might also be minor faults that might reasonably have escaped detection during product development. In these cases, the significance of the faults should be evaluated and the options for obtaining corrections might be pursued. However, if one or more serious faults pertaining to the product itself are discovered, confidence decreases rapidly regarding the suitability of the product for use in a safety-related application.

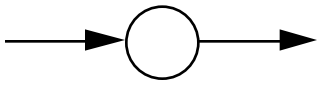
3.5. Discussion of Structural Testing

A brief summary of several structural testing methods is given here. The material in this section is based largely on Beizer 1990; see that reference for detailed tutorials. Note also that domain testing and logic testing (discussed in Section 4) are structural testing techniques if applied to a software object's implementation instead of to its specifications.

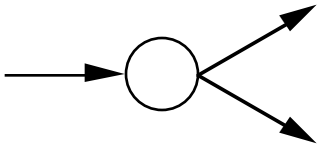
3.5.1. Control Flowgraphs

Structural testing methods generally make use of a control flowgraph of the module being tested. This is an abstraction of the module in the form of a directed graph which captures only the properties of the module which are being tested. Control flowgraphs are defined (informally) as follows:

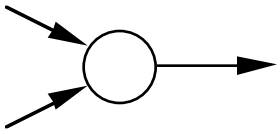
- A block of statements which do not involve control transfer in or out except from one to the next are replaced by a simple node of the control graph:



- A branch statement (IF statement) is replaced by a node representing the branch predicate with one edge for each outgoing branch:



- A junction is represented by a node with two incoming edges:



- A loop statement (DO, WHILE, FOR) is replaced by its component initialization–increment–test parts.

Figure 3-2 shows a (nonsensical) sample module. Figure 3-3 is the corresponding flowgraph. Each node is numbered; the numbers are repeated in the program listing in Figure 3-2. (They are for annotation only, not part of the module.)

```
        Begin module                              1
L2:     x = x + 1                                 2
        y = 7
L3:     if z < 4                                  3
                        then x = 2*x              4
                        else x = x – 1            5
        if z = 0        then go to L2             6
        if y = x - z    then go to L3             7
        if z = 2*z                                8
                        then y = y - 1            9
                        w = 2*x
        else if z < 2   then go to L2             10
        end module                                11
```

Figure 3-2. Example of a Program

A *link* is defined to be an edge of the flowgraph—it represents transfer of control from one block of code to another. A *segment* is a sequence of nodes and links—for example, in Figure 3-3, the progression through nodes 2, 3, 4, 6, and 7 is a segment. A *path* is any segment from the initial node of the module to the terminal node. The path contains a *loop* if any node is repeated. The *length* of the path is the number of links on the path. The *number of paths* is the number of distinct paths. For all but the simplest module, there are a large number of paths (a path with one iteration of a loop is distinct from the same path but with two loop iterations).

3.5.2. Control Flow (Path) Testing

Path testing is aimed at discovering software faults existing in the flow of control within a module; it does not address calculations within the module except for those calculations that affect the flow of control. It is assumed in this discussion that the module is written in a third-generation programming language (such as C, Pascal, or Ada), has a single entry point, and has a single exit point.[21] See Beizer 1990 for extensions to assembly language modules.

Execution of a module consists of the execution of a path within the module. Different inputs to the module may cause different paths to be executed. In the languages being considered, statements fall into several sets: arithmetic, branches, and loops. Branches usually consist of IF, CASE, GOTO and RETURN statements.[22] Loops usually consist of DO (or FOR) and WHILE statements, plus GOTOs that return control to a previously executed statement. Any path that contains the same statement more than once has a loop.

The completeness of control flow testing is referred to as test coverage. Two criteria for coverage measurements are statement coverage and branch coverage. Statement coverage requires that every statement in the module be executed at least once (also called node coverage). In Figure 3-3, 100% node coverage is achieved if all nodes are executed at least once. Since it is possible to cover all nodes without covering all links (for example, in Figure 3-3 a set of paths can be established to cover all nodes without traversing links such as 10–2 and 7–3), statement coverage is a very weak criterion that is never sufficient in safety-related applications.

Therefore statement coverage will not be discussed further. Branch coverage requires that the set of test cases cause every statement, every alternative of every branch, and every loop statement to be executed.

Methods (mostly heuristic) exist to create a reasonable number of test cases that include all statements and all branches (Beizer 1990). These are beyond the scope of this report. The module may need to be instrumented (modified by inserting code) so that evidence can be obtained to ensure that each test case performs as

[21] Note that multiple RETURN statements within the module do not constitute separate exit points, since they can be easily modified to single formal exit points without change in correctness.

[22] Syntax varies among languages—the forms used here are typical.

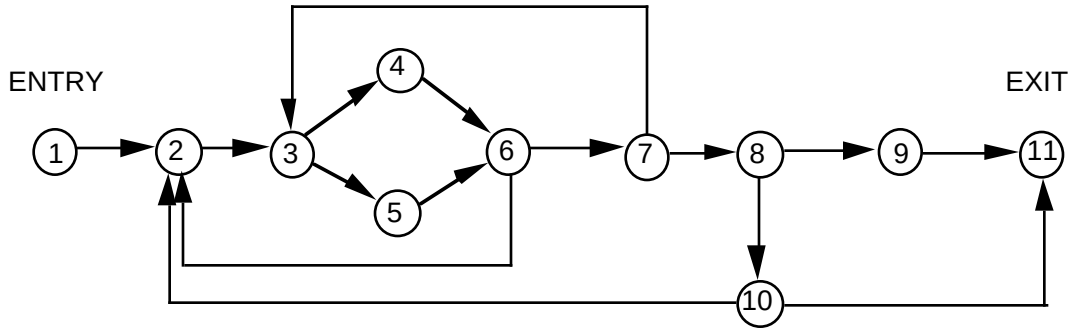


Figure 3-3. Flowgraph Corresponding to the Module in Figure 3-2

predicted. The internal state of the module (values of local variables) may also need to be examined to verify test case results.

3.5.3. Loop Testing

A loop occurs whenever a node is repeated in a path. This includes both well-structured and ill-structured loops; examples of both are shown in Figure 3-4.

Loop testing assumptions are similar to path testing assumptions. It is assumed that faults exist only in the flow of control around loops—there are no calculation faults or branch faults. It is assumed that the module specifications are correct and achievable. It is assumed that data used in the module is correctly defined and accessed.

Faults in loops tend to occur around the minimum and maximum number of iterations that are possible. Consider the loop statement

for n = 1 to k do {...}

where $1 \leq k_{min} \leq k \leq k_{max} < \infty$;
k_{min} and k_{max} fixed

(k_{min} and k_{max} are the minimum and maximum possible values of k.)

Loop testing consists of attempting to create test cases that force the numbers of iterations executed to take on values "near" both k_{min} and k_{max}, i.e., to create test cases that force the loop to execute typically one iteration more and less than either the lower limit, k_{min}, or the upper limit, k_{max}. In the example above, test cases should force "zero" iterations of the loop and compare actual results to the expected behavior of the software object (see Figure 3-5).In many cases, not all of these choices are possible. If the minimum number of iterations is zero ($k_{min} = 0$), one can hardly force $k_{min} - 1 = -1$ iterations. If there is no hard upper bound on k_{max}, the cases involving k_{max} are not possible.

Nested loops generally require all combinations of these choices for each nested loop. One loop has eight cases; two nested loops, 64; three nested loops, 512. Beizer suggests methods to reduce this number considerably, but such reductions should be subjected to a thorough analysis and used with great caution in safety-related applications.

Intertwining loops such as those shown in the bottom of Figure 3-4 require much more care and ingenuity to test adequately. It is far preferable to forbid the use of these loops if that option exists.

3.5.4. Data Flow Testing

Data flow testing is directed toward finding errors in the manipulation of data. This includes all types of data—program variables, sensor and actuator data, database data, and file data. Databases and files can be considered to be data, as well as the records in them.

Beizer defines four ways data can be manipulated. The exact meaning varies among the types of data. Symbols and meanings are shown in Table 3-1.

Data flow is analyzed by examining the way each data element in the module is used along each path in the control flowgraph. Sequences of the letters d, k, and u are used to record the ways the data is used. For example, consider data element x in the example of Figure 3-2. The variable x is manipulated in some way in each of the nodes 2, 4, 5, 7 and 9. In Figure 3-6, the usage of variable x is shown on the outlink of each node in which x is used in some way. Consider the usage of data element x on path 1-2-3-4-6-7-8-9-11 in Figure 3-6. The progression of usages of variable x on this path is represented by the string 'ududuu.' Note that if x is a local variable, an anomaly is indicated by the first 'ud' in that x is used before the first definition. This is shown in the second line of the example program in Fig. 3-2.

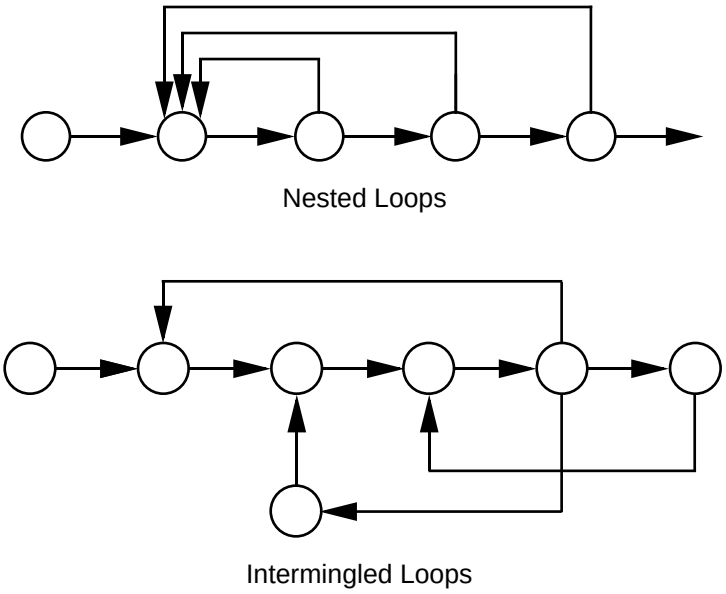


Figure 3-4. Examples of Loops in Flowgraphs

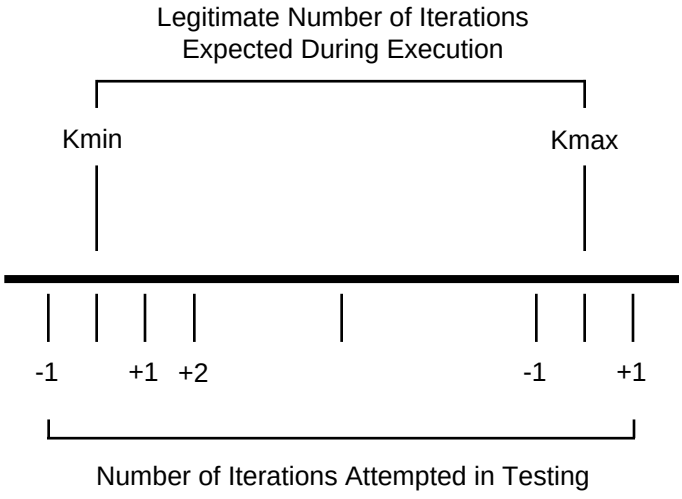


Figure 3-5. Test Cases in Loop Testing

Table 3-1. Data Flow Testing Symbols and Meanings

Symbol	Meaning	Definition
d	Defined	A program variable is defined when it is initialized in a declaration statement or is given a value in an assignment statement. A file is defined by being opened. A record is defined by being read.
k	Killed	A local variable is killed in block-structured programming languages when the containing procedure is exited. A file is killed when it is closed. A record in a file is killed when it is deleted. A record in a memory buffer is killed when the buffer is cleared.
c	Computation use	A program variable is used in a computation if it appears on the right-hand side of an assignment statement or as part of a pointer calculation.
p	Predicate use	A program variable is used in a predicate if it appears in the predicate portion of an IF, CASE, or WHILE statement.
u	Used	A variable is can be described as "used" if it is used in a computation (c) or in a predicate (p).

Analysis of data flow consists of examining the possible ways in which data may be used. Consideration is given to anomalous situations related to single usages of data items or to pairs of usages of a data item. Notation for single data item usages is of the form: -d, -u, -k, d-, u-, or k-, where the '-' means that nothing of interest regarding the data item is occurring on the path before or after the indicated usage type. Pairs of usages of a data item are indicated by a two-character string such as du or dk.

Fifteen combinations of single or paired usage are possible, and can be classified as follows:[23]

- The following combinations are considered normal: –d (the first definition on the path), k–, du, kd, ud, uk, and uu.
- The following combinations are suspicious, and should be investigated to be sure the usage is intended and is correct: –k, –u, d–, u–, dd, dk, and kk.
- The following combination is always a fault: ku.

Data flow testing requires the creation of test cases that cover these potential manipulations of data (i.e., that test the various calculations and variable usages). They are based on the module's control flowgraph, and should be considered as tests added to branch and loop tests. A number of such tests suites have been described in the literature, but are generally insufficient for safety-critical software. The recommended form is called the 'all-uses' strategy, and can be stated very simply:

> "There must be at least one test case for at least one path from every definition of every variable to every use of that definition."

Starting from the test cases already defined for branch and loop testing, consider the variables manipulated by the module one at a time. For each variable x, find all definitions of x and all uses ('c' or 'p') of x. For each definition, locate all 'c' and 'p' uses of that definition. For each such use, find a path on the control flowgraph from that definition to the use which does not include new definitions or kills. In many cases, an existing test case will be sufficient, since branch test cases and previous data flow test cases are likely to include the new data flow case.

In Figure 3-6, for variable 'x,' definitions occur in nodes 2, 4, and 5. The definition in node 2 is used in both nodes 4 and 5, while each of the definitions in nodes 4 and 5 are used in nodes 7 and 9. The latter two paths are included in the former, so only two paths are required here to cover all definitions/usage pairs: 1-2-3-4-6-7-8-9-11 and 1-2-3-5-6-7-8-9-11. In most cases, of course, many more paths would be required, including some which were not included in the control flow paths.

[23] Recall that ${}_nC_r = n!/[r!(n–r)!]$, so it can be shown that the following number of combinations is possible: ${}_5C_1 + {}_5C_2 = 5 + 10 = 15$ ways.

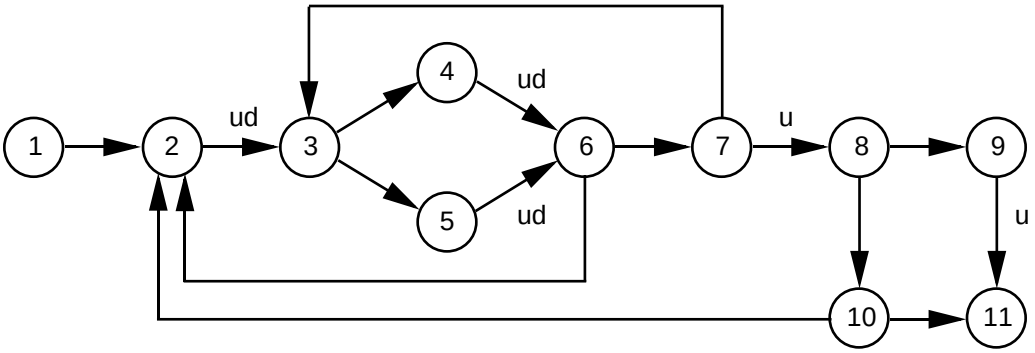


Figure 3-6. Control Flowgraph Augmented to Show Data Flow

4. FUNCTIONAL TESTING

4.1. Purpose of Functional Testing

Functional testing (also known as “black box” testing) consists of testing the functions to be performed by a software element as defined in requirements, design specifications and user documentation. It is focused on comparing actual and specified behavior, independent of the structural characteristics of the software. The primary concerns are functional and timing requirements. Programs, subsystems and systems are tested in large part with functional tests, however, functional tests also apply to packages and modules. Although, structural testing and static analyses are the dominant testing strategies for at these levels, the design specifications for packages and modules should contain information on which to base functional tests. This is particularly true for software elements such as communications packages, device drivers, and mathematical subroutines. For COTS software, functional testing is likely to be applied to programs, subsystems and systems, and will normally be carried out by or on behalf of the customer. The qualities addressed by functional testing, summarized in Table 1-2, are discussed below.

4.2. Benefits and Limitations of Functional Testing

Both the benefits and limitations of functional testing are a result of the fact that the execution of functions is examined rather than the internal structure of the software object. The focus is on verifying that requirements and user needs have been met. Functional testing can be applied at any level but is usually associated with programs, subsystems, and systems.

4.2.1. Benefits

Since the focus is not on internal software structure, it is easier for functional testing to be performed by independent parties. Test cases may originate with the customer, user, or regulator. For COTS software, test cases might also originate from information gathered from the experience of other users of the item. Finally, functional testing techniques do not require the availability of source code, which, for COTS software, may not be available.

Functional testing techniques can address a wide range of software qualities. Test cases for functional testing techniques address technical correctness by allowing verification of the accuracy and precision of results as well as verification of the consistency, interoperability, and performance of the software item. Consistency and interoperability are addressed by examining the interactions among modules as transactions are processed. Performance is addressed via test cases focused on timing requirements for real-world transactions. Regarding the correctness of a software item in the sense of its being complete, acceptable, and valid, test cases can focus on missing or partially implemented transactions, improper handling of real-world conditions and states, and incorrect representations of user needs and the real-world environment. Functional test cases can also be designed to test security and integrity mechanisms, user interfaces, and robustness in the presence of invalid inputs.

4.2.2. Limitations

Functional testing usually does not detect undocumented features or functions such as development aids left in the software. Since testers have no visibility into internals, functional subtleties may be overlooked, particularly if structural testing has not been performed.

4.3. Information Required to Perform Functional Testing

Functional testing requires a software requirements specification, user instructions, detailed knowledge of external interfaces (to sensors, actuators, operators, and other software), and the software object being tested.

A test station is recommended for testing by customers. This includes the ability to select pre-defined test cases, apply the test cases to the software object, and evaluate the results of the test against pre-defined criteria. The ability to reproduce functional testing will generally be necessary, and a test station is the most effective tool to accomplish this.

4.4. Methods of Performing Functional Testing

The functional testing must be planned, designed, created, executed, evaluated, and documented.

4.4.1. Test Planning and Test Requirements

The following actions are required to plan and generate requirements for functional testing.

1. Determine the software qualities that are being evaluated. For safety-related COTS software, the primary quality of interest is correctness. In a technical sense, this encompasses accuracy,

precision, consistency, interoperability, and performance. From a product perspective, correctness includes acceptability, completeness, and validity. Other qualities that can be addressed are integrity, security, robustness, usability, and user friendliness.

2. Determine which functional testing techniques will be required. Transaction testing and domain testing are minimal requirements. Additional techniques should be employed if the goal of the technique is applicable to the software object.

3. Determine what resources will be required in order to carry out the testing. Resources include budget, schedule, personnel, equipment, test tools, test station, and test data.

4. Determine the criteria to be used to decide how much testing will be required. This is a stopping criterion—how much testing is enough?

5. Determine which software objects will be tested.

4.4.2. Test Design and Test Implementation

The following actions are required to design and implement functional testing.

1. Create procedures for executing the functional test cases.

2. Create individual test cases. Each test case should contain the following information:
   a. *Test identification.* Each test case must have a unique identifier.

   b. *Purpose.* Each test case should have a specific reason for existing. Examples include verifying that a particular timing constraint can be met, that a particular function is performed correctly, or checking for a specific type of failure. For the latter, see headings 1 and 2 of the Bug Taxonomy in the Annex.

   c. *Input data.* The precise data required in order to initiate the test must be specified.

   d. *Initial state.* In order to reproduce a test case, the initial state of the software object (before the test begins) may need to be specified. This information is not necessary if the object is intended to execute correctly and identically in all initial states. For example, a transaction processing program should correctly handle any transaction no matter what has gone before.

   e. *Test results.* The expected results of the test must be known and specified. These are the values of data objects external to the software object under test (such as actuator values, display screen values, and database values).

   f. *Final state.* In some cases, the final state of the object must be specified as part of the test case information.

3. Create the test station. This is a mechanism for selecting, executing, evaluating, and recording the results of tests carried out on the object. Test station components, illustrated in Figure 4-1, include:
   a. *Test case selection*. A means of selecting test cases to be executed. Test case information is typically kept in a file or database, and the selection may simply consist of "get next test case."

   b. *Test program*. A means of setting the object's initial state (if necessary), providing input to the object, recording the output from the object, and (if necessary) recording the final state of the object.

   c. *Test oracle*. A means of determining the correctness of the actual test output and object state.

   d. *Results database*. A means of recording the test results for future analysis and evaluation. Typical data include the test identifier, date, version of object being tested, test output and state, and an indication of correctness or failure of the test.

4.4.3. Test Execution and Test Evaluation

The test procedures must be carried out and the results analyzed. If discrepancies between actual and expected results occur, there are two possibilities: either the test case has a fault or the object has a fault. In the first case, the test case should be corrected and the entire test procedure rerun.

If the object has a fault and the development organization is performing the test, the programmer is expected to correct the fault. The pattern of test–fix–test–fix continues until all discrepancies have been resolved.

In the case of COTS software, obtaining corrections may be very difficult. Suppose the test is being performed by (or on behalf of) the customer. If the software was developed for the customer under contract, there may be considerable leverage for obtaining corrections. If the software is a consumer product (for example, a library accompanying a

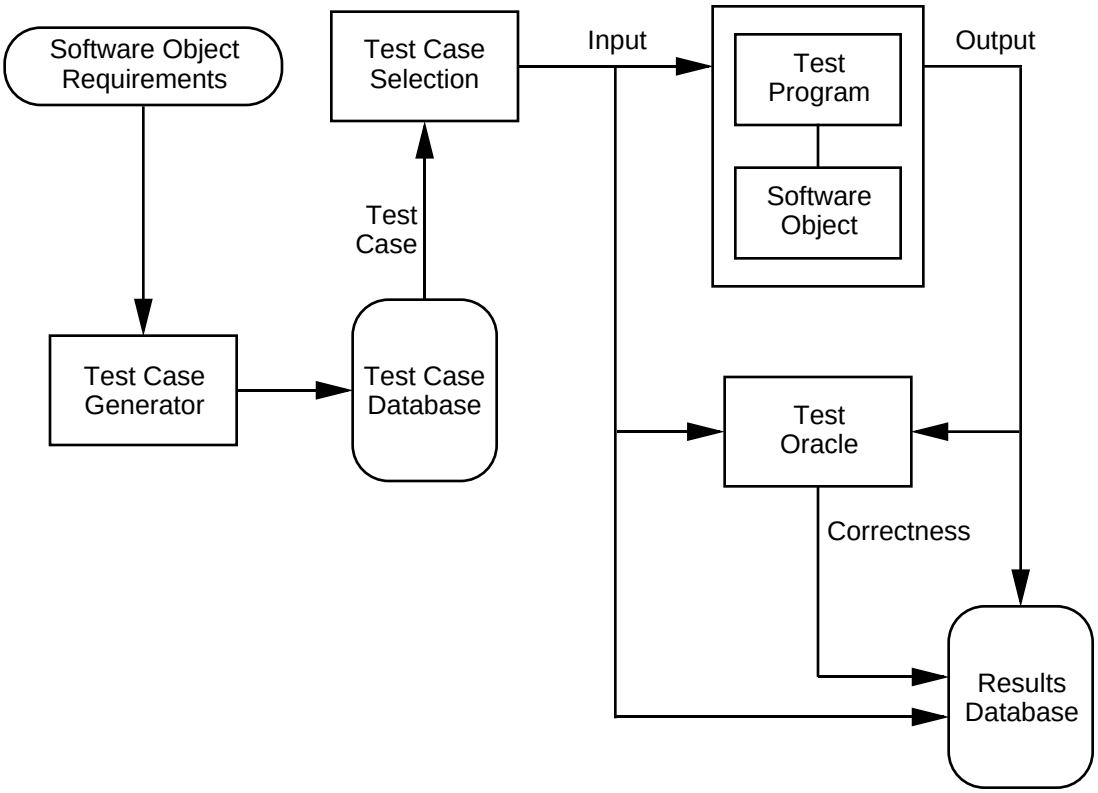


Figure 4-1. Typical Test Station Components for Functional Testing

compiler used for development), experience shows that many developers have little interest in expensive repairs that satisfy a limited marketplace. In this case, the options of the customer may be simply to reject the software or to evaluate each fault detected by the testing and determine its effects on safety. If more than one fault exists, the cumulative effect of all the faults on safety must also be determined.

The nature of the faults encountered must also be considered. The discovered faults might be related to new requirements arising from the specific, intended application of the COTS product. They might also be minor faults that might reasonably have escaped detection during product development. In these cases, the significance of the faults should be evaluated and the options for obtaining corrections might be pursued. However, if one or more serious faults pertaining to the product itself are discovered, confidence decreases rapidly regarding the suitability of the product for use in a safety-related application.

4.5. Discussion of Functional Testing

There are a number of techniques for developing functional tests. A summary of several of these is given below and is based largely on Beizer (1990); see that reference and Howden (1987) for detailed information.

4.5.1. Transaction Testing

A transaction is a complete unit of work as seen by the operators of the computer system. An example is changing the value of a set point. The operator normally views this as a simple action—entering a value on a display screen causes the value to change, resulting in a change to some other portion of the screen. In fact, many processes may be invoked on multiple computers to carry out the action, but none of the details are of interest to the operator.

Transaction testing is similar to control flow testing (Section 3.4.2) in that it is based on a flowgraph. It differs from control flow testing in that the flows in transaction testing are derived from the requirements specification, while the flows in control flow testing are derived from the program internal structure. Transaction testing is carried out at the program, subsystem, or system level instead of the module level. Nodes on the flowgraph represent processes that act on the transaction, while the links on the graph represent the movement of transaction data from one process to another. Note that a transaction flowgraph does not necessarily match program control flow. An example is shown in Figure 4-2.

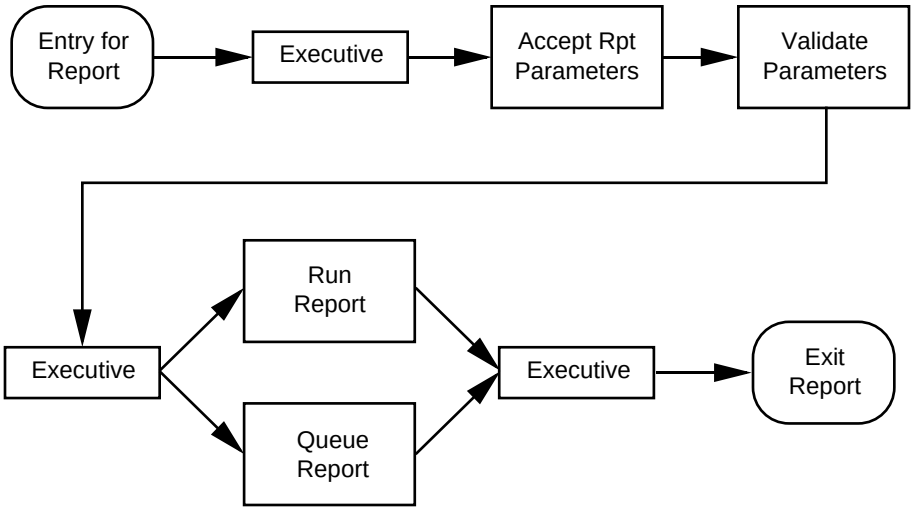


Figure 4-2. Example of a Transaction Flowgraph

Transactions typically are born (created) as a result of some triggering action, exist for some period of time, and then die. Each transaction can be modeled by a transaction flowgraph, and there is a separate graph for each transaction. Some computer systems involve hundreds of transactions, resulting in a large supply of graphs. Alternative flows on the graph may exist to handle errors and peculiar conditions. Transactions may "spawn" additional transactions, and multiple transactions may collapse into a single one. The resulting flow can be quite complex.

Transaction testing assumes that the processing within each node of the flowgraph is correct, but that there may be errors in routing transactions from one node to another. A test must be created for every path from transaction birth to transaction death. Particular attention must be devoted to paths caused by errors, anomalous data or timing, or other strange events.

4.5.2. Domain Testing

A program can frequently be viewed as a function transforming input values to output values. Programs generally operate on more than one input variable, and each adds a dimension to the input space. The collection of all input variables determines a vector, known as the input vector. An example might be

(temperature, pressure, neutron flux, on/off switch, valve position)

where the first three are assumed to be read from sensors and the last two read from an operator console. Domain testing divides the input vector values into sets, called domains, where the program behaves the same for all values in the set.

An example of a specification for a control function based on a single variable, temperature, might take the following form (the errors are deliberately included for illustrative purposes):

if temp < 0	error
if 0 < temp < 50	turn on heater
if 50 ≤ temp < 80	turn off both heater and cooler
if 75 < temp < 150	turn on cooler (this is assumed to be a specification error, i.e., assume the requirements call for shutdown at 120)
if 150 < temp	emergency shutdown (this is assumed to be a specification error, i.e., assume the requirements call for shutdown at 120)

In this example (illustrated in Figure 4-3), there are five domains, with boundaries at 0, 50, 80, and 120. The boundaries are typically points in the input space at which a new rule applies. The calculations are assumed to be correct for all values in each set, and faults are sought at or near the boundaries of the domain. Several errors are shown in the example:

- It is not known how the program should respond for temp = 0 and temp = 150.
- There are inconsistent requirements for 75 < temp < 80 since the domains overlap.
- The problem statement requires (it is assumed here) emergency shutdown at 120, not 150.

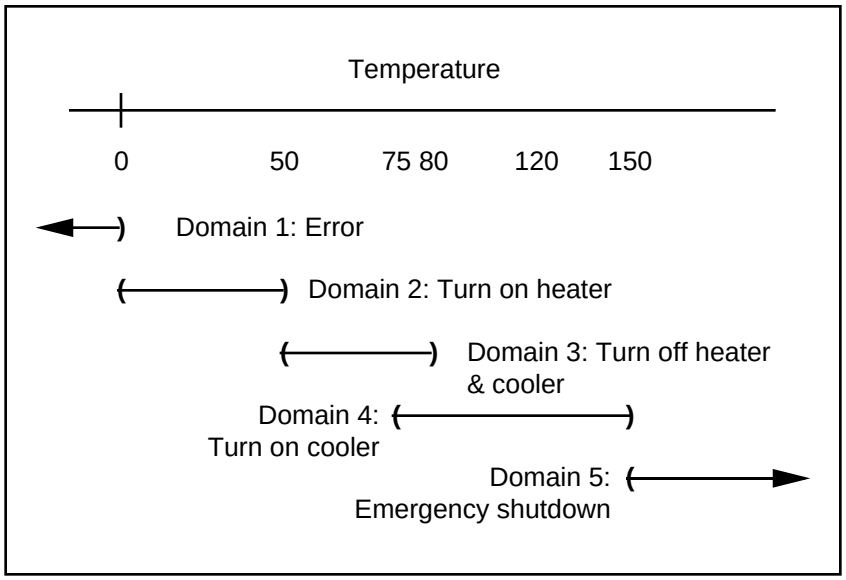


Figure 4-3. Example of Domains

Domains can be drawn and analyzed manually for one or two variables. Real-time control system software generally requires more than two sensors and operator signals, so the application of domain testing can be impractical unless automated tools can be found.[24]

Test cases for domain testing are concentrated at or very near the boundaries of each domain. Figure 4-4 shows hypothetical two-dimensional input spaces, where the shaded areas represent anticipated input values. The asterisks show a few of the possible test input values. If test cases are based on code implementation rather than specifications, domain testing is considered to be a structural technique.

Howden (1981) points out that techniques for examining classes of input data can also be applied to the examination of classes of output data. In cases where classes of output data are related to classes of input data, selecting input data to produce output at the boundaries of the output classes can yield useful results. In addition, it is also useful to consider invalid output data and to attempt to generate this output with selected inputs. This approach is closely related to the use of fault tree analysis.

4.5.3. Syntax Testing

The syntax of external inputs, such as operator or sensor inputs, and internal inputs, such as data crossing interfaces between subsystems, must be validated. In addition to the well-documented input syntax that may be described in the requirements and design specifications, it is also necessary to examine the software object for implicit, undeclared languages. These may be found in areas such as user and operator command sets, decision logic relating to transaction flows, and communications protocols. Sources for this information include requirements and design documentation, manuals, help screens, and developer interviews. Items relating to hidden languages should be included on code inspection checklists (see Section 2). For defined or hidden languages, the syntax must be defined with a tool such as BNF and a set of syntax graphs must be created on which to base test cases for various syntactic constructions. Figure 4-5 shows a trivial example of a syntax graph. A sentence would be formed based on the syntax graph by following a path indicated by the arrows, making legitimate substitutions when rectangles are encountered, and inserting literally the contents of the circles. Thus, PAUSE; and PAUSE{5}; would be legitimate constructions.

Testing consists of supplying a combination of valid and invalid constructions as inputs. Types of faults discovered with syntax testing relate to cases where valid constructions are not accepted, invalid constructions are accepted, or where the handling mechanisms for valid or invalid inputs break down. Beizer (1990) notes that the invalid constructions lead to the biggest payoffs in this type of testing. Fairly simple syntax rules can lead to very large numbers of possible test cases, so automated means must be used to accomplish the testing.

[24] A reviewer pointed out that he was unaware of the use of domain testing in real-time systems.

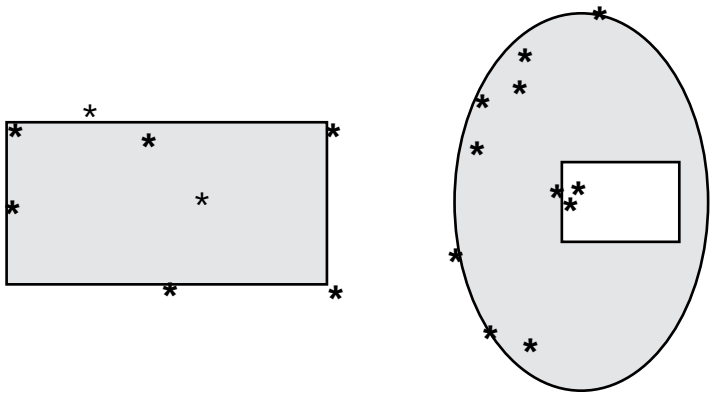


Figure 4-4. Examples of Two-Dimensional Domains with Examples of Test Values

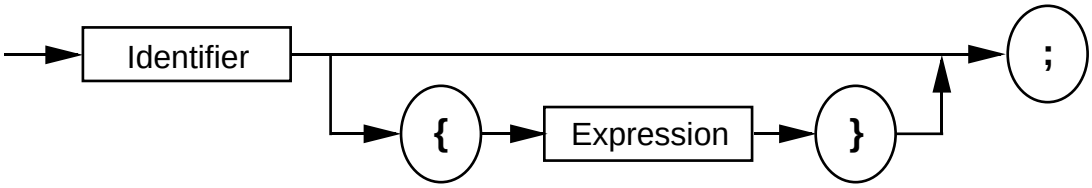


Figure 4-5. Example of a Syntax Graph

4.5.4. Logic-Based Testing

Some applications or implementations must deal with situations in which the values of a number of conditions must be evaluated and appropriate actions taken depending on the particular mix of condition values. If these situations are derived from system requirements, they are functional issues; if they are the result of the design approach, they are structural issues. Functional logic-based testing consists of testing the software system's logic for handling these mixes of conditions. In addition to the correctness of the logic, software quality factors of completeness and internal consistency are also addressed.

Decision tables can be an effective means for designing test cases to examine software logic. This logic might be explicitly documented using techniques such as decision tables or decision trees, or might be implicit in the software requirements or design specifications. In the latter case, the sources for obtaining information are the same as for syntax testing. The cause–effect graphing technique can be applied to transform this information into decision table format; an example is provided in Pressman (1987).

An example of a limited entry (conditions and actions are binary valued) decision table is shown in Figure 4-6. A detailed discussion of decision tables can be found in Hurley (1983). A rule consists of the actions to be followed when the specified conditions hold. Note that the rule corresponding to conditions (Y,Y,Y) is missing, possibly corresponding to an impossible physical situation. The dash in rule 4 means that the value of condition 3 is immaterial for this rule (i.e., rule 4 represents two cases, N,Y,Y and N,Y,N).

Testing based on this decision table should begin with a verifying the completeness and consistency of the table (see Hurley, 1983). Then test cases should be developed to ensure that the software performs the correct actions for the specified rules. It should be verified, by attempting to design a test case, that a (Y,Y,Y) situation is indeed impossible, and both options for rule 4 should be tested to ensure that the same action is taken.

	RULES					
	1	2	3	4	5	6
Condition 1	Y	Y	Y	N	N	N
Condition 2	Y	N	N	Y	N	N
Condition 3	N	Y	N	-	Y	N
Action 1					X	
Action 2	X			X		X
Action 3		X				X
Action 4			X	X	X	X

Figure 4-6. Example of a Decision Table

The use of a decision table model for designing tests is appropriate when the following requirements hold (Beizer 1990):

- The specification consists of, or is amenable to, a decision table .
- The order of condition evaluation does not affect rule interpretation or resulting actions.
- The order of rule evaluation does not affect resulting actions.
- Once a rule is satisfied, no other rule need be considered.
- If multiple actions can result from a given rule, the order in which the actions are executed does not matter.

4.5.5. State Testing

Testing based on state-transition models is effective in examining a number of areas including communication protocols, failure and recovery sequences, and concurrent processing. Figure 4-7 illustrates a state transition diagram with three states indicated by boxes and three transitions indicated by arrows. The trigger for the state change (input or event) is shown in the top part of the transition label and the action or output associated with the transition is shown in the bottom part of the label. (Note that state-transition models can be depicted with other notation, such as state tables.) For each input to a state, there must be exactly one transition specified; if the state doesn't change, a transition is shown to and from the same state.

Faults can be associated with an incorrect structure for a state-transition model or with a structurally correct model that does not accurately represent the modeled phenomena. In the former category, faults can be related to conditions such as states that cannot be reached or exited or the failure to specify exactly one transition for each input. These types of faults can be detected from a structural analysis of the model. In the latter category, faults can be related to conditions such as states missing from the model, errors in the specification of triggering events, or incorrect transitions. Detection of these errors involves the analysis of, or testing against, specifications. Missing states can arise from incorrect developer assumptions about possible system states or real world events. Errors in modeling triggering events or associated outputs can easily arise from ambiguities contained in system or software requirements. For embedded COTS software, states of the software itself or states related to the interface of the software to the larger system may need to be modeled as a basis for analysis and testing.

To perform state testing, it is first necessary to develop correct state-transition diagrams for the phenomena being investigated. An analysis should be made to verify that the state-transition model is consistent with the design and that the model to be used is structurally correct. Design errors might be indicated by this analysis. Following this analysis, a set of test cases should be developed that, as a minimum, covers all nodes and links of the diagrams. Test cases should specify input sequences, transitions and next states, and output sequences.

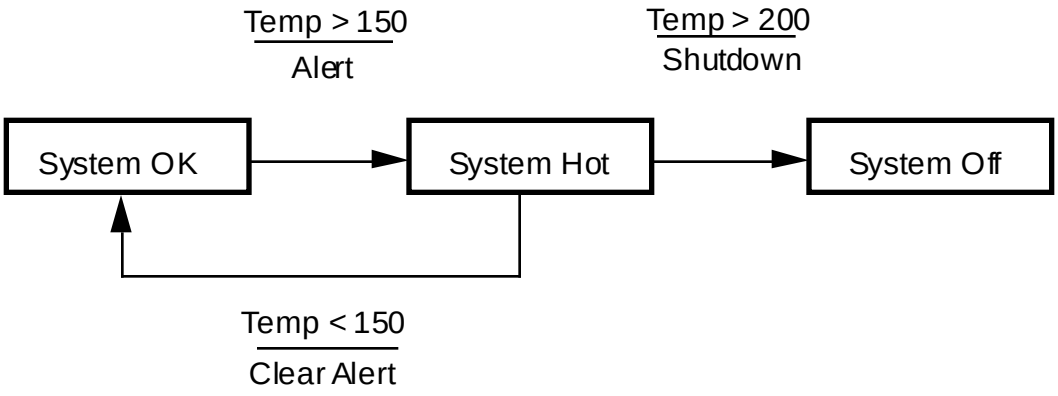


Figure 4-7. Example of a State Transition Diagram

State testing is recommended in the following situations (see Beizer 1990):

- Where an output is based on the occurrence of sequences of events
- Where protocols are involved
- Where device drivers are used
- Where transactions can stay in the system indefinitely
- Where system resource utilization is of interest
- Where functions have been implemented with state-transition tables
- Where system behavior is dependent upon stored state.

5. STATISTICAL TESTING

5.1. Purpose of Statistical Testing

Statistical testing is conducted to measure the reliability of a software object or to predict its probability of failure, rather than to discover software faults. It consists of randomly choosing a sample of input values for the software object and then determining the correctness of the outputs generated from those inputs. Obtaining a statistically valid reliability measure using this testing strategy requires that the following assumptions hold:

1. The test runs are independent.
2. For each input, the chance of failure is constant. That is, the probability of failure is independent of the order in which samples are presented to the software object, and of the number of samples that precede the specific input.
3. The number of test runs is large.
4. All failures during testing are detected.
5. The distribution of the inputs under real operating conditions is known.

The qualities addressed by statistical testing are availability and reliability.

It is possible to use statistical testing for the goal of finding failures (random testing). That is, one runs randomly selected tests in the hopes of finding failures. This is likely to be less efficient than the other, more directed, forms of testing. Of course, if failures do happen during statistical testing, the faults should be found and corrected. See Hamlet (1994) for a discussion of random testing.

5.2. Benefits and Limitations of Statistical Testing

5.2.1. Benefits

Statistical testing does not rely on any knowledge of the internal composition of the software object, so it can be carried out whether or not such knowledge exists. It is the only way to provide assurance that a specified reliability level has been achieved. Statistical testing (as discussed here) is less prone to human bias errors than other forms of testing. It is a practical method in many cases when moderate-to-high reliability (in the range of 10^{-4} to 10^{-5} failures per demand) is required.

Statistical testing addresses the reliability quality by estimating probabilities based on large numbers of tests. Reliability information also provides information regarding potential availability, although it does not address external factors, such as system loads or administrative procedures, that may affect accessibility when a particular capability is needed.

5.2.2. Limitations

A number of practical issues with statistical testing limit its usefulness in some instances. The first set of issues relates to the test planning and test station (see below). The most difficult of these issues are frequently the construction and verification of the test oracle. Determining the operational profile may be nearly as difficult.

The second set of issues involves the length of time necessary for testing. Testing to the level of reliability required for a typical safety-critical process control system should be feasible, but testing to much higher levels of reliability is not. (See the discussion of expected test duration in section 5.4.2.)

The third set of issues concerns the relationship between safety and reliability. Statistical testing provides a reliability number, not a safety number. Since inputs with safety implications should be a very small percentage of all possible inputs, it is not likely that random testing will include many safety-critical input cases. In such cases, it may be possible to carry out two series of tests: one based on all possible input cases, and one based only on safety-critical input cases. This would result in two numbers—an overall reliability figure and a safety-related reliability figure. The latter could be reasonably termed a safety reliability number. This approach does, however, require that the set of safety-critical input events be completely understood so that the safety-critical input space can be completely and accurately characterized. This may be difficult to accomplish.

5.3. Information Required to Perform Statistical Testing

Statistical testing requires no knowledge of the internal composition or structure of the software object being tested. It does require a good understanding of the statistical distribution of inputs which can be expected to appear during actual operating conditions (the operational profile). A test platform is required, which includes the ability to generate random tests using the operational profile, the ability to carry out each test on the software object, and the ability to evaluate the

results for correctness. Since many thousands of tests are required in order to obtain a valid reliability number, the test platform must be automated.

5.4. Methods of Performing Statistical Testing

The statistical test must be planned, designed, implemented, executed, evaluated, and documented. The following steps (or their equivalent) must be carried out.

5.4.1. Test Planning and Test Requirements

The following actions are required to plan and generate requirements for statistical testing. Statistical testing focuses on the reliability quality of software. For safety-related COTS software, the goal of statistical testing is to provide a measure of the item's reliability given its anticipated operational profile (i.e., given the specific role that the COTS item will play in the safety-related system). The software qualities of interest for statistical testing are reliability and availability.

1. Determine the level of reliability to be achieved. This is generally given in terms of the maximum acceptable failure rate—for example, that the failure rate cannot exceed 10^{-5} per demand.

2. Determine if failures will be tolerated. A statistical test will be carried out for some period of time, recording all failures. At some point, the number of failures may be so large that the test will be stopped and the software object rejected. If the test is to be statistically valid, this point must be determined during test planning. For reactor protection systems, the objective should be to carry out the test without failure. In this case, any failure will cause the test to stop, the fault to be corrected, and the test to be completely rerun. When a statistical test is re-run, it is crucial that the random numbers selected be independent of sequences previously used.

3. Determine the degree of statistical confidence which will be required in the test results. This will be given as a percentage—for example, .99.

4. Determine what resources will be required in order to carry out the testing. Resources include budget, schedule, personnel, equipment, test tools, test station, and test data.

5. Determine which software objects will be tested.

5.4.2. Test Design and Test Implementation

The following actions are required to design and implement statistical testing.

1. Calculate the number of test cases which must be carried out without failure to achieve the specified reliability with the specified confidence level.

The number of test cases, n, is given by the following formula, where f is the failure rate and c is the confidence level (Poore, Mills, and Mutchler 1993):

$$n = \left\lceil \frac{\log(1-c)}{\log(1-f)} \right\rceil$$

Table 5-1 shows approximate values of n for various values of c and f. In this table, 'M' stands for 'million.' This table shows that increasing the required level of confidence in the test results can be obtained with relatively little extra effort. However increasing the required level of reliability (decreasing the failure rate) that must be demonstrated requires considerably more test cases to be executed and consequently increases test time.

Given a required number of test cases and an assumption about the average number of test cases that can be carried out per unit time, estimates can be made of the total time that will be required for test execution. Table 5-2 shows the approximate amount of execution time required to achieve specified failure rates at the .99 confidence level under two assumptions of the rate of testing: one test per second and one test per minute. In the first case, testing is impractical for failure rates under 10^{-7}; in the latter, under 10^{-5}. Note that this table assumes that tests are carried out 24 hours per day, seven days per week, and that no failures are encountered during the test. Determining the expected amount of calendar (elapsed) time for the test will be longer if the assumptions are not valid. The times given in the table are examples; if test cases require more (or less) time, then the table can be adjusted. For example, if a test case requires five minutes to execute, then nearly six years will be required for a failure rate of 10^{-5}.

Table 5-1. Required Number of Test Cases to Achieve Stated Levels of Failure Rate and Confidence

f	c=.9	c=.99	c=.999
10^{-1}	22	44	66
10^{-2}	230	460	690
10^{-3}	2,300	4,600	6,900
10^{-4}	23,000	46,000	69,000
10^{-5}	230,000	460,000	690,000
10^{-6}	2,300,000	4,600,000	6,900,000
10^{-7}	23M	46M	69M
10^{-8}	230M	460M	690M
10^{-9}	2,300M	4,600M	6,900M
10^{-10}	23,000M	46,000M	69,000M
10^{-11}	230,000M	460,000M	690,000M

Table 5-2. Expected Test Duration as a Function of Test Case Duration

Failure rate	Number of test cases	1 test per second	1 test per minute
10^{-1}	44	44 seconds	45 minutes
10^{-2}	459	7.5 minutes	7.6 hours
10^{-3}	4600	1.25 hours	3 days
10^{-4}	46,000	13 hours	1 month
10^{-5}	460,000	5.5 days	11 months
10^{-6}	4,600,000	1.75 months	9 years
10^{-7}	46M	1.5 years	90 years
10^{-8}	460M	15 years	900 years
10^{-9}	4,600M	150 years	9,000 years
10^{-10}	46,000M	1,500 years	90,000 years
10^{-11}	460,000M	15,000 years	900,000 years

2. Obtain the operational profile.

An operational profile is a statistical distribution function which gives, for every point *p* in the input space, the probability that *p* will be selected at any arbitrary point in time. More formally, suppose the inputs presented to the software object during actual operation are v_1, v_2, ..., v_n. Then the operational profile gives, for each point *p*, the probability that v_k = p for each k, $1 \le k \le n$ (Musa 1992)[25].

For example, suppose that a software object has only three input values: low, medium, and high. An analysis of the expected frequency of these three values shows that 'low' will occur 70% of the time; 'medium,' 20%; and 'high,' 10%. This is an operational profile for this example.

3. Determine the test oracle.

This is a function which, given an input to the software object under test and the results of running the test, will determine whether the actual test result obtained is correct. The test oracle must be able to make this determination with very high confidence.

[25] Some additional statistical assumptions discussed in the reference are not listed here.

4. Create the test station.

A test station is a mechanism for creating, executing, evaluating, and recording tests performed on the software object and the results of the tests. It must be able to run with minimal supervision for very long periods of time. Typical test station components are shown in Figure 5-1.

A brief description of each component of a test station follows:

a. *Input Generator.* A means of generating input test cases in such a way that the probability distribution function of the test cases is equivalent to the probability distribution function determined by the operational profile.

b. *Test Program.* A means by which the software object can be executed using the generated test cases as input to produce test results as output. As a general rule, the object must be placed in the same initial state before each test is carried out.

c. *Test Oracle.* A means of determining the correctness of the output produced by the software object under test.

d. *Test Database.* A means of recording the test input, test output, and correctness for future analysis and evaluation.

5.4.3. Test Execution and Test Evaluation

The following actions are required to execute and evaluate statistical testing.

1. *Execute the tests.* Carry out the test procedure until the predetermined number of test cases have been executed without failure. The number of test cases which will be required can be determined from Table 5-1.

2. *Assess the tests.* Evaluate the results to be sure that the test was successfully executed, and provide assurance of this fact. This may require a formal certification.

5.5. Discussion of Statistical Testing

Statistical testing is the primary way to calculate a failure rate for a software object. When the conditions discussed above can be met, statistical testing can be very effective. It can be used for nearly any type of software object.

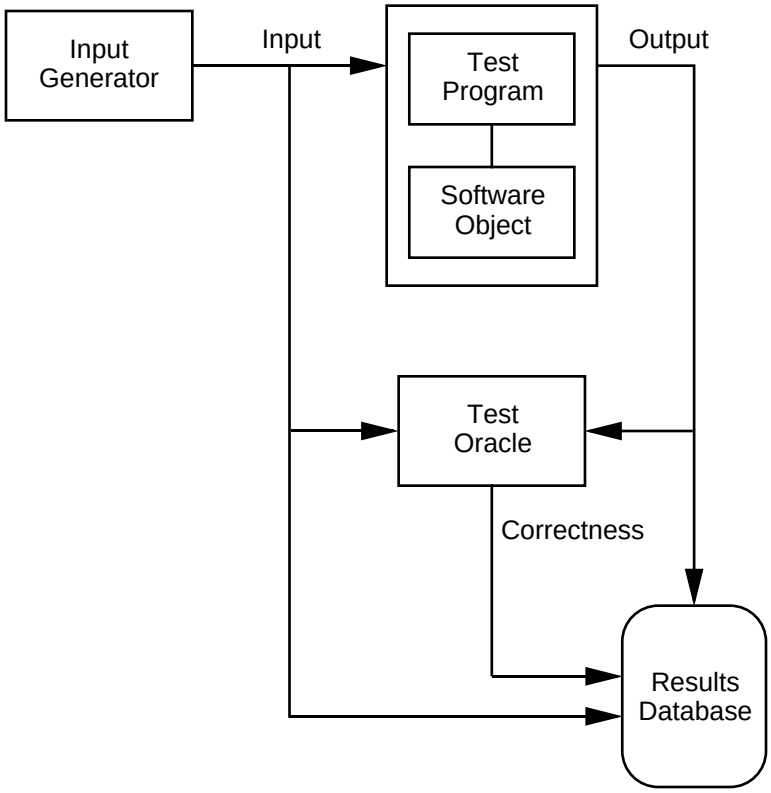


Figure 5-1. Typical Test Station Components for Statistical Testing

For example, suppose it is necessary to provide a reliability number for a square root routine. It would be reasonable to assume that the operational profile function is the uniform distribution function, so that all random numbers are equally likely to be used. Generating a sequence of random numbers for this distribution is easy, so the input generator is simply a random-number generator. The test program merely calls the square root routine. The oracle is simple—check for a positive number, square the answer and compare to the input number using previously established error bounds. It should be possible to carry out one test every millisecond or so, depending on the speed of the computer being used. If the goal is a failure rate of 10^{-8} with .99 confidence, Table 5-1 shows that about 460,000,000 test cases will be required—this will take about 5.3 days.

Statistical testing will be much more difficult for a software system such as a reactor protection system. Here, the input points may consist of a series of values from simulated sensors which occur over a period of several minutes—and the timing may be critical. This would mean that carrying out a sequence of tests will require a considerable amount of time. Assuming one test per minute (on average), attaining a failure rate of 10^{-4} at .99 confidence will require about a month of testing. This is estimated as follows:

1. Table 5-1 states that approximately 46,000 test cases are required to achieve a failure rate of 10^{-4} at .99 confidence level.
2. The assumption of one test case executing per minute (on average) means that sixty test cases can be executed in an hour. Assuming that the tests are automated and run continuously 24 hours a day, seven days a week, it follows that 10,080 test cases can be executed in a calendar week.
3. Hence, it will require (46,000)/(10,080) or approximately 4.5 calendar weeks to execute the required test cases to establish this statistical failure rate at the specified confidence level.

Similarly, it can be shown that attaining a failure rate of 10^{-5} will require nearly a year of testing.

An accurate operational profile may be difficult to obtain. One possible approach is to partition the input space into subsets of inputs that occur in different modes of operation, and test each of these individually, assuming a uniform distribution function. For example, one mode of operation could be “all operating parameters well within bounds;” another could be “some operating parameter is near a limit,” and so on. If these operational modes can, in turn, be specified accurately, statistical testing can be carried out for each mode. (See Whittaker 1994 for an alternative approach.)

There are some advantages to this approach. It is presumably more important to know the reliability of the software under off-normal and emergency conditions than under normal operating conditions. One might be willing to test for 10^{-4} failure rate under normal conditions, but require 10^{-5} under near-emergency and emergency conditions. If the latter input space is sufficiently small, increased confidence in the software could be obtained at reasonable cost.

However, constructing the test oracle and guaranteeing its correctness becomes a serious problem. It is not possible to carry out large numbers of tests and evaluate the results using human labor because of the time constraints and human error rates for this type of task.

Appendix B

6. STRESS TESTING

6.1. Purpose of Stress Testing

Stress testing is a process of subjecting a system to abnormal loads on resources in order to discover whether the system can function outside anticipated normal ranges, to determine the usage limits beyond which the system will fail as a result of the overloaded resource, and to gain information that will help to characterize the behavior of a system when it is operating near its usage limits. The process of discovering "breaking points" also provides the opportunity to examine recovery mechanisms and procedures.

If a system can function adequately with loads outside the anticipated real-life application domain levels, the assumption is that it will perform properly with normal loads (Perry 1988). Background testing (testing in the presence of loads within normal ranges) should be performed to help validate this assumption. A background test verifies that the system will perform adequately within the normal mix of loads and resources and provides the basis with which to compare stress test results.

Stress testing is particularly important for COTS software items since those items may not have been developed with the particular safety-related application in mind. This type of testing provides an opportunity to examine the COTS software performance with respect to the intended application.

The qualities addressed by stress testing, summarized in Table 1-2, are discussed below.

6.2. Benefits and Limitations of Stress Testing

6.2.1. Benefits

Stress testing forces a system to operate in unusual circumstances not typically created in other forms of testing and, therefore, is complementary to other elements of the overall testing effort. It is particularly important for safety-related software since it is a testing strategy that creates high-stress, off-normal scenarios in which the software is likely to fail. For reactor protection systems, these scenarios might be related to sensor input streams of interrupt-type or buffer loading signals or to output streams generated in emergency situations. Stress testing uncovers information about software faults and provides an understanding of limits on system resources. The latter is useful in validating the intended use of the COTS item, in establishing system monitoring routines, and in tuning the system for installed operations.

Stress testing provides information about robustness and performance by creating scenarios in which normal operating ranges are exceeded and examining how performance degrades. Stress testing at the boundaries of these ranges also allows one to confirm that performance requirements have been met. The actual failures encountered in stress testing may lead to the discovery of software faults and provide opportunities to examine the completeness of the recovery mechanisms incorporated into the software.

6.2.2. Limitations

Stress testing must be performed in an actual or simulated installed environment and requires complete information about operating and user procedures. Stress testing can be expensive because of manpower costs or because of the need to develop automated elements of the test station. In addition, specific internal states can be difficult to reproduce, and root causes of failures can be difficult to find.

6.3. Information Required to Perform Stress Testing

Stress testing must be performed in an actual or simulated production (installed) environment. Therefore, complete information about this environment must be available, including an understanding of operating and user procedures. Since stress testing must provide abnormal loads, there must be a definition of the types of loads to be placed on the system as well as an understanding of what the normal operating ranges will be for each load. Typical load types of interest are as follows (the first four being of particular interest for reactor protection systems):

- High volumes and arrival rates of transactions
- Saturation of communications and processor capacities
- Situations stressing internal table sizes
- Situations stressing internal sequencing or scheduling operations
- Heavy use of disk storage space and swapping capability
- Operating with a very large database size
- Many simultaneous users.

Finally, if available, design information is valuable in order to understand how to design specific stress tests that will focus on internals.

6.4. Methods of Performing Stress Testing

The stress tests must be planned, designed, created, coordinated, executed, evaluated, and documented.

6.4.1. Test Planning and Test Requirements

The following actions are required to plan and generate requirements for stress testing.

1. Determine the software qualities to be addressed with stress testing. The primary quality of interest for safety-related COTS software is robustness in the intended role; however, availability, completeness, correctness, and performance are also addressed.
2. Determine the load situations under which the software system is to be tested. For safety-related COTS software, these will be determined based on knowledge of the role that the COTS product will play in the system and vendor-supplied information regarding product functions and performance. Information derived from the usage experience of other users of the COTS software item or from fault tree analyses of the system is also valuable in this process.
3. Determine whether the stress testing environment will be an actual or simulated production environment.
4. Determine the resources required to carry out the testing. Resources include budget, schedule, personnel, equipment, test tools, test station, and test data.
5. Determine the criteria to be used to decide how much testing will be required. This is a stopping criterion—how much testing is enough? For example, "stress testing of a particular software resource might continue until adequate information has been gathered regarding all three goals of stress testing."

6.4.2. Test Design and Test Implementation

The following actions are required to design and implement stress testing.

1. Establish the testing environment for the stress tests.

In most cases, a simulated production environment will be required. Since the results of stress testing will reflect the performance of the software in the test environment rather than the real-life environment, the simulated production environment should be as close as possible to the actual production environment.

2. Create procedures for executing the stress tests.

Since this testing will take place in an actual or simulated production environment, the test procedures should make use of system operating procedures and usage procedures or user guides. The stress test procedures specify how the system loads will be generated, the roles of all participants, the sequences of operations (scripts) each participant will perform, the test cases to be performed, and how the test results will be logged.

3. Create individual test cases.

Each test case should contain the following information:

a. *Test identification.* Each test case must have a unique identifier.

b. *Purpose.* Each test case should have a specific reason for existing. Examples include verifying the proper operation of a system function, verifying response times, and verifying the handling of exception conditions during situations of high system loads.

c. *Input data.* The precise data required in order to initiate the test case must be specified.

d. *Initial state.* The initial state for the test case is essentially specified in the test procedures and scripts; however, there may be initial state information specific to a given test case.

e. *Test results.* The expected results of the test must be known and specified. Expected performance statistics, counts of operations, etc., should be determined from the planned load and test case input data.

f. *Final state.* In some cases, the final state is important and must be specified as part of the test case information.

4. Create the test station.

The test station is a mechanism for specifying and generating loads as well as selecting, executing, evaluating, and recording the results of other tests carried out on the software. Note that, depending on the goal of a particular stress (or background) test, input may consist solely of transactions in the input load or may be augmented by test cases from other types of testing. Test station components (patterned after Beizer 1984) are illustrated in Figure 6-1 and include:

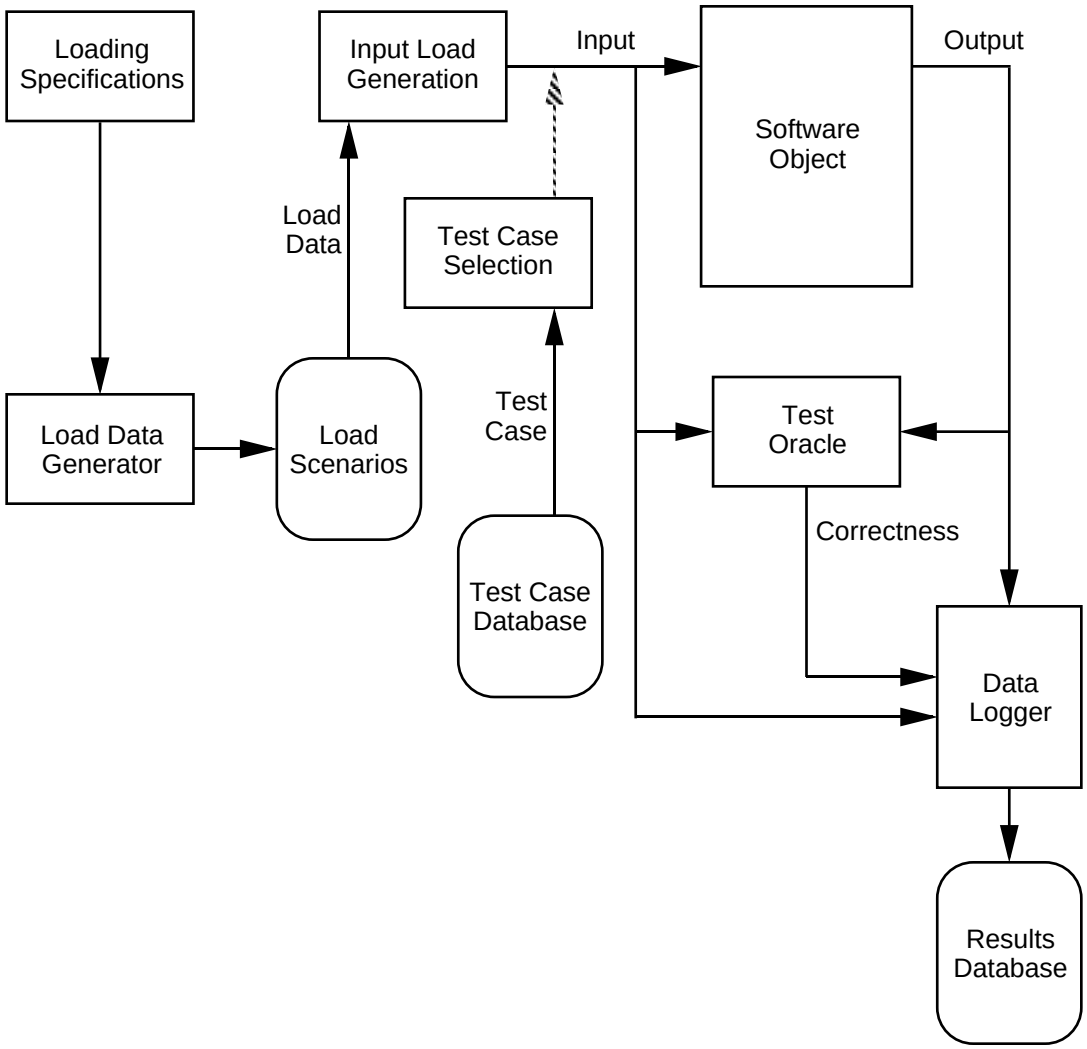


Figure 6-1. Typical Test Station Components

a. *Load data generator.* A means of accepting specifications for the loading of resources and generating scenarios needed for the input load generator to produce the required load during the stress test run.

b. *Test case selection.* A means of selecting, if appropriate, additional test cases to be executed. Test case information is typically kept in a file or database, and the selection may simply consist of "get next test case."

c. *Input load generation.* A means of accepting input data for loading and generating the desired system loads with the desired statistical characteristics.

d. *Test Oracle.* A means of determining the correctness of the output (of the optional test cases) produced by the software object under test.

e. *Data Logger.* A means of logging pertinent information about system performance during the stress test.

f. *Test evaluation.* A means of analyzing the results of the stress test, including specific test case results as well as scanning software system output for anomalies created during the stress test.

g. *Results database.* A means of recording the test results for future analysis and evaluation.

6.4.3. Test Execution and Test Evaluation

The test procedures must be carried out and the test results must be analyzed. The logged system outputs, produced in response to the input load or any additional test cases, must be examined to verify correct operation of the system. This is done by comparing inputs and outputs relating to specific test cases or transactions to determine if information was

lost or improperly processed. In addition, the logged output must be analyzed to determine if timing, sequencing, counts, error recovery, etc. match what was input to the system by the load generator and test case selector. If appropriate, database integrity checking routines should be run. If a particular load causes the system to fail, the logged information is used to search for the circumstances of the failure and to quantify the load level at which the failure occurred. These evaluations can be quite difficult to perform since they can require the careful examination of voluminous data.

The results of stress testing may indicate that the system performs acceptably within the planned load ranges of the tested resources. In this case, the stress testing results provide operating information about resource limits that can then be embedded into system monitoring routines or used for system tuning purposes.

The performance profile and the nature of faults encountered must also be considered. The performance information must be verified against the requirements and constraints of the system in which the COTS software will operate. The significance of the faults discovered should be evaluated and, if appropriate, the options for obtaining corrections might be pursued. If the performance of the software is not within the requirements of the application or if one or more serious faults are discovered, confidence decreases rapidly regarding the suitability of the product for use in a safety-related application.

6.5. Discussion of Stress Testing

Stress testing is a process of subjecting a system to abnormal loads with the intention of breaking the system. By investigating why the system breaks and at what load level the system breaks, information is gained about possible software faults, operating load limits, system behavior near the load limits, and error recovery behavior. Typical software faults discovered are faults associated with sequencing, contention for resources, scheduling priorities, error or time-out recovery paths, and simultaneous activities. These faults tend to be subtle and frequently indicate design problems rather than simple coding mistakes (Beizer 1984).

For COTS software, there are a number of approaches to identifying specific load situations to test. The role of COTS software in the overall system must be characterized with respect to functions provided, performance requirements, and interfaces to other elements of the system—essentially a black-box characterization. Additional information can be added based on any available vendor data regarding product specification and target performance levels. If source code is available, the source code inspection process (see Section 2) could have, as one of its goals, a focus on identifying structural properties that should be stress tested. Information can also be gathered from other users of the COTS software item regarding usage experience and load ranges. This information might suggest suspect areas or provide additional confidence that some areas are robust. Finally, if fault tree analysis techniques are applied to the overall system, any root causes possibly relating to the COTS software role must be examined to see if load-related failures might be possible.

With respect to the task of diagnosing software faults based on stress test results, it should be noted that the exact reproduction of internal states resulting from stress test scenarios is difficult, if not impossible. This is because the simultaneous activities of test participants and the various internal timing and scheduling situations are usually not exactly repeatable. Therefore, the process of identifying software faults based on stress test results is not as deterministic as it is for other types of test results analysis. However, if the goal is to examine general behavior at various load levels, the stored scenarios can be re-run as needed. Discovered software failures that are reproducible without active system loads can be further investigated with other test techniques. It is more difficult to diagnose software failures that occur only under high system loads or that cannot be reproduced in subsequent stress tests. For this reason, it is important to have full knowledge of test inputs and to log as much information as possible during the stress test execution for subsequent analysis. For COTS software items, knowledge of the experience of other users and a characterization of their normal operating loads is useful ancillary information for analysis of test results.

Creating mechanisms for generating the required loads, logging test data, and analyzing results is a difficult task. For small systems with minimal real-time requirements or in cases in which only general information such as user response time is desired, it is possible to do the data generation manually and to create the system load via interactive user inputs augmented by other system functions such as running reports and performing intense data searches. Data logging would be done manually or automatically using existing logging features, and test results analysis would be manual. Even though there might not be a need for developing automated load generators in these cases, there will still be a significant effort to use and coordinate manpower and system resources for stress testing.

For most situations, it is necessary to develop automated means for generating the input loads, logging data, and analyzing results. See Beizer 1984 for a detailed discussion of load generation techniques.

The load generation process comprises two parts, which can be combined or separated depending on the demands of the stress testing operations. First, the information characterizing a particular load is used to generate typical test data according to statistical distributions of desired input parameters. Second, an automated means for using this data to generate loads in real time during the test must be created. Logging facilities might already exist in the system platform; if not, they have to be created. Finally, the analysis of results will require specialized routines to organize and summarize the data, scan the results for possible anomalies, and compare system performance statistics with those anticipated from the input load statistics.

Appendix B

7. REGRESSION TESTING

7.1. Purpose of Regression Testing

Regression testing consists of re-running a standard set of test cases following the implementation of a set of one or more changes to previously tested software. Its purpose is to provide confidence that modifications have not had unintended effects on the behavior of the software. It is assumed that the appropriate testing techniques (see the other sections of this report) have been applied to test whether the modified software elements perform as specified in the change documentation. In addition to regression testing itself, it is necessary to verify that all system documentation, such as requirements, design, and operating procedures, have been updated to reflect the software modifications. Regression testing addresses the quality of software correctness and, indirectly, the qualities associated with the test strategies that are being re-applied.

7.2. Benefits and Limitations of Regression Testing

7.2.1. Benefits

In addition to the direct testing of software modifications, regression testing is required to provide assurance that, except for the modified portion, the software performs in the same way that it did prior to the changes. Since the regression testing process repeats previous testing, no additional "start-up" costs are associated with establishing test mechanisms. In addition, since the regression testing effort is largely the same for each software change, there is benefit in combining changes into one release. For COTS software, this is equivalent to determining when to upgrade to a new release.

The primary software quality of interest in regression testing is correctness since the goal is to verify that new faults have not been inadvertently introduced into the software. Since regression testing consists of re-running test cases from appropriate test techniques, the qualities associated with those techniques are also re-examined during regression testing.

7.2.2. Limitations

There are significant maintenance costs for configuration management of the test cases, test data, and test procedures as well as for keeping the testing environment(s) current. Regression testing will involve re-running large numbers of test cases in a variety of types of testing and will, therefore, be expensive to perform.

7.3. Information Required to Perform Regression Testing

Since regression testing is a re-use of existing test cases,[26] the information required to perform this testing depends upon the specific types of test cases to be re-run. This information is described in the sections of this report dealing with the test types of interest. It is essential that configuration control be maintained on all test documentation and related test materials to permit regression testing to be performed effectively and efficiently.

7.4. Methods of Performing Regression Testing

The regression tests must be planned, designed, coordinated, executed, evaluated, and documented.

7.4.1. Test Planning and Test Requirements

The following actions are required to plan and generate requirements for regression testing.

1. Establish and maintain the standard set of tests (test cases, data, and procedures) to be repeated as regression tests. For COTS software used in a safety-related context, it is recommended that the full set of functional and stress testing initially conducted be repeated.

2. Determine what resources will be required in order to carry out the testing. Resources include budget, schedule, personnel, equipment, test tools, test station(s), and test data. The test tools, test station(s), and test data should already be in place from previous testing activity, and should be directly usable provided that configuration management procedures have been continuously applied to these items.

3. Determine the criteria to be used to decide how much testing will be required. This is a stopping criterion—how much testing is enough? For example, "the regression testing process will continue until the entire standard set of test cases runs without incident."

7.4.2. Test Design and Test Implementation

The following actions are required to design and implement regression testing.

[26] As a system evolves, the suite of test cases used for regression testing must also evolve.

1. Ensure that the testing environments used in previous testing have been maintained and are ready for regression testing.
2. Ensure that the modified software elements have been tested according to the same testing plans used on the original software.
3. Review the standard set of regression test cases, data, and procedures to discover whether any have been invalidated as a result of the desired modifications. Update the test cases and procedures as appropriate.

7.4.3. Test Execution and Test Evaluation

The test procedures must be carried out and the test results analyzed. Since the modified software elements have already been tested to verify correct operation, the regression test results should indicate that the areas of the COTS software thought to have been unaffected by the modifications are indeed unaffected by the changes. The results should exactly match the results of previous, successful tests.

7.5. Discussion of Regression Testing

The primary focus of regression testing is to provide assurance that implemented changes do not, in some subtle way, ripple through the system and cause unintended effects. In addition to software function and performance, there must be a verification that conventions, standards, access rules, etc., were adhered to in the change implementation. One source of problems occurring in software maintenance is that undocumented assumptions made by the development team are not carried over into the maintenance phase (Hetzel 1984). For these reasons, it is recommended that the full complement of functional and stress testing activities originally performed be repeated to test the modified safety-related software system. (It is assumed that appropriate tests and analyses will already have been run on the modified code.) Depending on the role of the modified software element and the criticality of its function (and of the overall software system), it may be possible to justify a reduced set of test cases for regression testing based on a change impact assessment and knowledge of potential fault consequences derived from a software risk assessment. This requires a careful assessment of the modified software element and its interfaces (logical and data) to other parts of the system, as well as a complete understanding of the likelihood and magnitudes of potential loss.

The methods used for regression testing are the same methods used for the various types of testing carried out previously. Test plans, test cases, and test procedures, as well as test stations and automated test support mechanisms, already exist, and it is assumed that they have been maintained under configuration control for future use in regression testing. Whenever modifications are made to the software object, it is necessary to review the standard set of test cases (as well as test data and test procedures) to ensure that none have been invalidated by the modifications and to update the set based on the specifications for the newly modified object.

8. REFERENCES

Basili, Victor R. and Richard W. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering* Vol. 12, No. 12 (December 1987), 1278–1296.

Beizer, Boris, *Software System Testing and Quality Assurance*, Van Nostrand Reinhold (1984).

Beizer, Boris, *Software Testing Techniques*, Van Nostrand Reinhold (1990).

Charette, Robert N., *Software Engineering Risk Analysis and Management*, McGraw-Hill (1989).

Dyer, Michael, *The Cleanroom Approach to Quality Software Development*, John Wiley & Sons (1992).

Fagan, M. E., "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal*, Vol. 15, No. 3, 1976, 182–211.

Glass, Robert L., *Building Quality Software*, Prentice-Hall (1992).

Hamlet, Richard, "Random Testing," in *Encyclopedia of Software Engineering*, John Wiley & Sons, 1994.

Hetzel, William, *The Complete Guide to Software Testing*, QED Information Sciences, Inc. (1984).

Howden, William E., "A Survey of Static Analysis Methods," in *Tutorial: Software Testing & Validation Techniques*, Institute of Electrical and Electronics Engineers, 1981.

Howden, William E., "A Survey of Dynamic Analysis Methods," in *Tutorial: Software Testing & Validation Techniques*, Institute of Electrical and Electronics Engineers, 1981.

Howden, William E., *Functional Program Testing and Analysis*; McGraw-Hill (1987).

Hurley, Richard B., *Decision Tables in Software Engineering,* Van Nostrand Reinhold, (1983).

IEEE 610.12. *IEEE Standard Glossary of Software Engineering Terminology*, Institute of Electrical and Electronics Engineers, 1991.

IEEE 829. *IEEE Standard for Software Test Documentation*, Institute of Electrical and Electronics Engineers, 1983.

IEEE 1008. *IEEE Standard for Software Unit Testing*, Institute of Electrical and Electronics Engineers, 1986.

IEEE 1074. *IEEE Standard for Developing Software Life Cycle Processes*, Institute of Electrical and Electronics Engineers, 1992.

Lawrence, J. Dennis, *Software Reliability and Safety in Nuclear Reactor Protection Systems*, NUREG/CR-6101, Lawrence Livermore National Laboratory, Livermore, CA (1993).

Lawrence, J. Dennis and Preckshot, G. G., *Design Factors for Safety-Critical Software,* NUREG/CR-6294, Lawrence Livermore National Laboratory, Livermore, CA (1994).

Marick, Brian, *The Craft of Software Testing*; Prentice-Hall (1995).

Miller, Edward and William E. Howden, *Tutorial: Software Testing and Validation Techniques*, Second Edition, IEEE Computer Society Press (1981).

McCall, Jim A. et al., "Factors in Software Quality," *Concept and Definitions of Software Quality*, General Electric Company, 1977.

Musa, John D., "The Operational Profile in Software Reliability Engineering: An Overview," Third Int'l Symp. on Soft. Rel. Eng. (October 1992), 140–154.

Olender, Kurt M. and Leon J. Osterweil, "Cecil: A Sequencing Constraint Language for Automatic Static Analysis Generation," *IEEE Transactions on Software Engineering*, Vol. 16, No. 3, March 1990, 268–280.

Perry, William E., *A Structured Approach to Systems Testing*, QED Information Sciences (1988).

Pressman, Roger S., *Software Engineering, A Practitioner's Approach,* McGraw-Hill, (1987).

Price, *Source Code Static Analysis Tools Report*, Software Technology Support Center, 1992.

Poore, J. H., Harlan D. Mills, and David Mutchler, "Planning and Certifying Software System Reliability," *IEEE Software 10*, 1 (January 1993), 88–99.

Preckshot, G. G. and Scott, J. A., *Vendor Assessment and Software Plans,* UCRL-ID-122243, Lawrence Livermore National Laboratory, Livermore, CA (1995).

Whittaker, James A., and Michael G. Thomason, "A Markov chain model for statistical software testing," *IEEE Transactions on Software Engineering,* Vol. 20, No. 10 (October 1994), 812-824.

Yourdon, Edward, *Structured Walkthroughs*, Prentice-Hall (1989).

Appendix B

ANNEX: TAXONOMY OF SOFTWARE BUGS

This Annex* provides a taxonomy for program faults (bugs). Faults are categorized by a four-digit number, perhaps with sub-numbers using the point system: e.g., "1234.5.6." The "x" that appears is a place holder for possible future filling in of numbers as the taxonomy is expanded. For example,

3xxx—structural bugs in the implemented software

32xx—processing bugs

322x—expression evaluation

3222—arithmetic expressions

3222.1—wrong operator

1xxx: FUNCTIONAL BUGS: REQUIREMENTS AND FEATURES: Bugs having to do with requirements as specified or as implemented.

11xx: REQUIREMENTS INCORRECT: the requirement or a part of it is incorrect.

111x: Incorrect: requirement is wrong.

112x: Undesirable: requirement is correct as stated but it is not desirable.

113x: Not needed: requirement is not needed.

12xx: LOGIC: the requirement is illogical or unreasonable.

121x: Illogical: illogical, usually because of a self-contradiction which can be exposed by a logical analysis of cases.

122x: Unreasonable: logical and consistent but unreasonable with respect to the environment and/or budgetary and time constraints.

123x: Unachievable: requirement fundamentally impossible or cannot be achieved under existing constraints.

124x: Inconsistent, incompatible: requirement is inconsistent with other requirements or with the environment.

1242: Internal: the inconsistency is evident within the specified component.

1244: External: the inconsistency is with external (to the component) components or the environment.

1248: Configuration sensitivity: the incompatibility is with one or more configurations (hardware, software, operating system) in which the component is expected to work.

13xx: COMPLETENESS: the requirement as specified is either ambiguous, incomplete, or overly specified.

131x: Incomplete: the specification is incomplete; cases, features, variations or attributes are not specified and therefore not implemented.

132x: Missing, unspecified: the entire requirement is missing.

133x: Duplicated, overlapped: specified requirement totally or partially overlaps another requirement either already implemented or specified elsewhere.

134x: Overly generalized: requirement as specified is correct and consistent but is overly generalized (e.g., too powerful) for the application.

137x: Not downward compatible: requirement as specified will mean that objects created or manipulated by prior versions can either not be processed by this version or will be incorrectly processed.

138x: Insufficiently extendible: requirement as specified cannot be expanded in ways that are likely to be needed—important hooks are left out of specification.

---

* This Annex is based on "Bug Taxonomy and Statistics," Appendix, *Software Testing Techniques*, second edition, by Boris Beizer. Copyright © 1990 by Boris Beizer. Reprinted with permission of Van Nostrand Reinhold, New York.

14xx: VERIFIABILITY: specification bugs having to do with verifying that the requirement was correctly or incorrectly implemented.

141x: Unverifiable: the requirement, if implemented, cannot be verified by any means or within available time and budget. For example, it is possible to design a test, but the outcome of the test cannot be verified as correct or incorrect.

142x: Untestable: it is not possible to design and/or execute tests that will verify the requirement. Untestable is stronger than unverifiable.

15xx: PRESENTATION: bugs in the presentation or documentation of requirements. The requirements are presumed to be correct, but the form in which they are presented is not. This can be important for test design automation systems, which demand specific formats.

152x: Presentation, documentation: general presentation, documentation, format, media, etc.

153x: Standards: presentation violates standards for requirements.

16xx: REQUIREMENT CHANGES: requirements, whether or not correct, have been changed between the time programming started and testing ended.

162x Features: requirement changes concerned with features.

1621: Feature added: a new feature has been added.

1632: Feature deleted: previously required feature deleted.

1633: Feature changed: significant changes to feature, other than changes in cases.

163x: Cases: cases within a feature have been changed. Feature itself is not significantly modified except for cases.

1631: Cases added.

1632: Cases deleted.

1633: Cases changed: processing or treatment of specific case(s) changed.

164x: Domain changes: input data domain modified: e.g., boundary changes, closure, treatment.

165x: User messages and diagnostics: changes in text, content, or conditions under which user prompts, warning, error messages, etc. are produced.

166x: Internal interfaces: direct interfaces (e.g., via data structures) have been changed.

167x: External interfaces: external interfaces, such as device drivers, protocols, etc. have been changed.

168x: Performance and timing: changes to performance requirements (e.g., throughput) and/or timings.

2xxx: FUNCTIONALITY AS IMPLEMENTED: requirement known or assumed to be correct, implementable, and testable, but implement is wrong.

21xx: CORRECTNESS: having to do with the correctness of the implementation.

211x: Feature misunderstood, wrong: feature as implemented is not correct—not as specified.

218x: Feature interactions: feature is correctly implemented by itself, but has incorrect interactions with other features, or specified or implied interaction is incorrectly handled.

22xx: COMPLETENESS, FEATURES: having to do with the completeness with which features are implemented.

221x: Missing feature: an entire feature is missing.

222x: Unspecified feature: a feature not specified has been implemented.

223x: Duplicated, overlapped feature: feature as implemented supplicates or overlaps features implemented by other parts of the software.

23xx: COMPLETENESS, CASES: having to do with the completeness of cases within features.

231x: Missing case.

232x: Extra case: cases that should not have been handled are handled.

233x: Duplicated, overlapped case: duplicated handling of cases or partial overlap with other cases.

234x: Extraneous output data: data not required are output

24xx: DOMAINS: processing case or feature depends on a combination of input values. A domain bug exists if the wrong processing is executed for the selected input-value combination.

241x: Domain misunderstood, wrong: misunderstanding of the size, shape, boundaries, or other characteristics of the specified input domain for the feature or case. Most bugs related to handling extreme cases are domain bugs.

242x: Boundary locations: the values or expressions that define a domain boundary are wrong: e.g., "X>=6" instead of "X>=3."

243x: Boundary closures: end points and boundaries of the domain are incorrectly associated with an adjacent domain: e.g., "X>=0" instead of "X>0".

244x: Boundary intersections: domain boundaries are defined by a relation between domain control variables. That relation, as implemented, is incorrect: e.g., "IF X>0 AND Y>0..." instead of "IF X>0 OR Y>0...".

25xx: USER MESSAGES AND DIAGNOSTICS: user prompt or printout or the form of communication is incorrect. Processing is assumed to be correct: e.g., false warning, failure to warn, wrong message, spelling, formats.

26xx: EXCEPTION CONDITIONS MISHANDLED: exception conditions such as illogical, resource problems, failure modes, which require special handling, are not correctly handled or the wrong exception-handling mechanisms are used.

3xxx: STUCTURAL BUGS: bugs related to the component's structure: i.e., the code.

31xx: CONTROL FLOW AND SEQUENCING: bugs specifically related to the control flow of the program or the order and extent to which things are done, as distinct from what is done.

311x: General structure: general bugs related to component structure.

3112: Unachievable path: a functionally meaningful processing path in the code for which there is no combination of input values that will force the path to be executed. Do not confuse with unreachable code. The code in question might be reached by some other path.

3114: Unreachable code: code for which there is no combination of input values that will cause that code to be executed.

3116: Dead-end code: code segments that once entered cannot be exited, even though it was intended that an exit be possible.

312x: Control logic and predicates: the path taken through a program is directed by control flow predicates (e.g., Boolean expressions). This category addresses the implementation of such predicates.

3122: Duplicated logic: control logic that should appear only once is inadvertently duplicated in whole or in part.

3124: Don't care: improper handling of cases for which what is to be done does not matter either because the case is impossible or because it really does not matter: e.g., incorrectly assuming that the case is a don't-care case, failure to do case validation, not invoking the correct exception handler, improper logic simplification to take advantage of such cases.

3126: Illogicals: improper identification of, or processing of, illogical or impossible conditions. An illogical is stronger than a don't care. Illogicals usually mean that something bad has happened and that recovery is needed. Examples of bugs include: illogical not really so, failure to recognize illogical, invoking wrong handler, improper simplification of control logic to take advantage of the case.

3128: Other control-flow predicate bugs: control-flow problems that can be directly attributed to the incorrect formulation of a control flow predicate: e.g., "IF A>B THEN ..." instead of "IF A<B THEN ...".

313x: Case selection bug: simple bugs in case selections, such as improperly formulated case selection expression. GOTO list, or bug in assigned GOTO.

314x: Loops and iteration: bugs having to do with the control of loops.

3141: Initial value: iteration value wrong: e.g., "FOR 13 TO 17 ..." instead of "FOR 1=8 TO 17."

3142: Terminal value or condition: value, variable, or expression used to control loop termination is incorrect: e.g., "FOR I = 1 TO 7 ..." instead of "FOR I = 1 TO 8."

3143: Increment value: value, variable, or expression used to control loop increment value is incorrect: e.g., "FOR I = 1 TO 7 STEP 2 ..." instead of "FOR I = 1 TO 7 STEP 5 ...".

3144: Iteration variable processing: where end points and/or increments are controlled by values calculated within the loop's scope, a bug in such calculations.

3148: Exception exit condition: where specified values or conditions or relations between variables force an abnormal exit to the loop, either incorrect processing of such conditions or incorrect exit mechanism invoked.

315x: Control initialization and/or state: bugs having to do with how the program's control flow is initialized and changes of state that affect the control flow: e.g., switches.

3152: Control initialization: initializing to the wrong state or failure to initialize.

3154: Control state: for state-determined control flows, incorrect transition to a new state from the current state: e.g., input condition X requires a transition to state B, given that the program is in state A; instead, the transition is to state C. Most incorrect GOTOs are included in this category.

316x: Incorrect exception handling: any incorrect invocation of a control-flow exception handler not previously categorized.

32xx: PROCESSING: bug related to processing under the assumption that the control flow is correct.

321x: Algorithmic, fundamental: inappropriate or incorrect algorithm selected, but implemented correctly: e.g., using an incorrect approximation, using a shortcut string search algorithm that assumes string characteristics that may not apply.

322x: Expression evaluation: bugs having to do with the way arithmetic, Boolean, string, and other expressions are evaluated.

3222: Arithmetic: bugs related to evaluated of arithmetic expression.

3222.1: Operator: wrong arithmetic operator or function used.

3222.2: Parentheses: syntactically correct bug in placement of parentheses or other arithmetic delimiters.

3222.3: Sign: bug in use of sign.

3224: Logical or Boolean, not control: bug in the manipulation or evaluation of Boolean expression that are not (directly) part of control-flow predicates: e.g., using wrong mask, AND instead of OR, incorrect simplification of Boolean function.

3226: String manipulation: bug in string manipulation.

3226.1: Beheading: the beginning of a string is cut off when it should not have been, or not cut off when it should have been.

3226.2: Curtailing: as for beheading but for string end.

3226.3: Concatenation order: strings are concatenated in wrong order or concatenated when they should not be.

3326.3.1: Append instead of precede.

3226.3.2: Precede instead of append.

3226.4: Inserting: having to do with the insertion of one string into another.

3226.5: Converting case: case conversion (upper to lower, say) is incorrect.

3226.6: Code conversion: string is converted to another code incorrectly or not converted when it should be.

3226.7: Packing, unpacking: strings are incorrectly packed or unpacked.

3228: Symbolic, algebraic: bugs in symbolic processing of algebraic expressions.

323x: Initialization: bugs in initialization of variables, expressions, functions, etc. used in processing, excluding initialization bugs associated with declarations and data statements and loop initialization.

324x: Cleanup: incorrect handling of cleanup of temporary data areas, registers, states, etc. associated with processing.

325x: Precision, accuracy: insufficient or excessive precision, insufficient accuracy, and other bugs related to number representation system used.

326x: Execution time: excessive (usually) execution time for processing component.

4xxx: DATA: bugs in the definition, structure, or use of data.

41xx: DATA DEFINITION, STRUCTURE, DECLARATION: bugs in this definition, structure, and initialization of data: e.g., in DATA statements. This category applies whether the object is declared statically in source code or created dynamically.

411x: Type: the data object type, as declared, is incorrect: e.g., integer instead of floating, short instead of long, pointer instead of integer, array instead of scalar, incorrect user-defined type.

412x: Dimension: for arrays and other objects that have a dimension (e.g., arrays, records, files) by which component objects can be indexed, a bug in the dimension, in the minimum or maximum dimensions, or in redimensioning statements.

413x: Initial, default values: bugs in the assigned initial values of the object (e.g., in DATA statements), selection of incorrect default values, or failure to supply a default value if needed.

414x: Duplication and aliases: bugs related to the incorrect duplication or failure to create a duplicated object.

4142: Duplicated: duplicated definition of an object where allowed by the syntax.

4144: Aliases: object is known by one or more aliases but specified alias is incorrect: object not aliased when it should have been.

415x: Scope: the scope, partition, or components to which the object applies is incorrectly specified.

4152: Local should be global: a locally defined object (e.g., within the scope of a specific component) should have been specified more globally (e.g., in COMMON)

4154: Global should be local: the scope of an object is too global: it should have been declared more locally.

4156: Global/local inconsistency or conflict: a syntactically acceptable conflict between a local and/or global declaration of an object (e.g., incorrect COMMON).

416x: Static/dynamic resources: related to the declaration of static and dynamically allocated resources.

4162: Should be static resource: resource is defined as a dynamically allocated object but should have been static (e.g., permanent).

4164: Should be dynamic resource: resource is defined as static but should have been declared as dynamic.

4166: Insufficient resources, space: number of specified resources is insufficient or there is insufficient space (e.g., main memory, cache, registers, disc) to hold the declared resources.

4168: Data overlay bug: data objects are to be overlaid but there is a bug in the specification of the overlay areas.

42xx: DATA ACCESS AND HANDLING: having to do with access and manipulation of data objects that are presumed to be correctly defined.

421x: Type: bugs having to do with the object type.

4212: Wrong type: object type is incorrect for required processing: e.g., multiplying two strings.

4314: Type transformation: object undergoes incorrect type transformation: e.g., integer to floating, pointer to integer, specified type transformation is not allowed, required type transformation not done. Note: type transformation bugs can exist in any language, whether or not it is strongly typed, whether or not there are user-defined types.

4216: Scaling, units: scaling or units (semantic) associated with objects is incorrect, incorrectly transformed or not transformed: e.g., FOOT-POUNDS to STONE-FURLONGS.

422x: Dimension: for dynamically variable dimensions of a dimensioned object, a bug in the dimension: e.g., dynamic redimension of arrays, exceeding maximum file length, removing one or more than the minimum number of records.

423x: Value: having to do with the value of data objects or parts thereof.

4232: Initialization: initialization or default value of object is incorrect. Not to be confused with initialization and default bugs in declarations. This is a dynamic initialization bug.

4234: Constant value: incorrect constant value for an object: e.g., a constant in an expression.

424x: Duplication and aliases: bugs in dynamic (run time) duplication and aliasing of objects.

4242: Object already exists: Attempt to create an object that already exists.

4244: No such object: attempted reference to an object that does not exist.

426x: Resources: having to do with dynamically allocated resources and resource pools, in whatever memory media they exist: main, cache, disc, bulk RAM. Included are queue blocks, control blocks, buffer blocks, heaps, files.

4262: No such resource: reference resource does not exist.

4264: Wrong resource type: wrong resource type reference.

428x: Access: having to do with access of objects as distinct from the manipulation of objects. In this context, accesses include read, write, modify, and (in some instances) create and destroy.

4281: Wrong object accessed: incorrect object accessed: e.g., "X:=ABC33" instead of "X:=ABD33".

4282: Access rights violation: access rights are controlled by attributes associated with the caller and the object. For example, some callers can only read the object, others can read and modify. Violations of object access rights are included in this category whether or not a formal access rights mechanism exits: that is, access rights could be specified by programming conventions rather than by software.

4283: Data-flow anomaly: data-flow anomalies involve the sequence of accesses to an object: e.g., reading or initializing an object before it has been created, or creating and than not using.

4284: Interlock bug: where objects are in simultaneous use by more than one caller, interlocks and synchronization mechanisms may be used to ensure that all data are current and changed by only one caller at a time. These are not bugs in the interlock or synchronization mechanism but in the use of that mechanism.

4285: Saving or protecting bug: application requires that the object be saved or otherwise protected in different program states or, alternatively, not protected. These bugs are related to the incorrect usage of such protection mechanisms or procedures.

4286: Restoration bug: application requires that a previously saved object be restored prior to processing: e.g., POP the stack, restore registers after interrupt. This category includes bugs in the incorrect restoration of data objects and not bugs in the implementation of the restoration of data objects and not bugs in the implementation of the restoration mechanism.

4287: Access mode, direct/indirect: object is accessed by wrong means: e.g., direct access of an object for which indirect access is required: call by value instead of name, or vice versa: indexed instead of sequential, or vice versa.

4288: Object boundary or structure: access to object is partly correct, but the object structure and its boundaries are handled incorrectly: e.g., fetching 8 characters of a string instead of 7, mishandling word boundaries, getting too much or too little of an object.

5xxx: IMPLEMENTATION: bugs having to do with the implementation of the software. Some of these, such as standards and documentation, may not affect the actual workings of the software. They are included in the bug taxonomy because of their impact on maintenance.

51xx: CODING AND TYPOGRAPHICAL: bugs that can be clearly attributed to simple coding, as well as typographical bugs. Classification of a bug into this category is subjective. If a programmer believed that the correct variable, say, was "ABCD" instead of "ABCE", than it would be classified as a 4281 bug (wrong object accessed). Conversely, if E was changed to D because of a typewriting bug, then it belongs here.

511x: Coding wild card, typographical: all bugs that can be reasonably attributed to typing and other typographical bugs.

512x: Instruction, construct misunderstood: all bugs that can be reasonably attributed to a misunderstanding of an instruction's operation or HOL statement's action.

52xx: STANDARDS VIOLATION: bugs having to do with violating or misunderstanding the applicable programming standards and conventions. The software is assumed to work properly.

521x: Structure violations: violations concerning control-flow structure, organization of the software, etc.

5212: Control flow: violations of control-flow structure conventions: e.g., excessive IF-THEN-ELSE nesting, not using CASE statements where required, not following dictated processing order, jumping into or out of loops, jumping into or out of decisions.

5214: Complexity: violation of maximum (usually) or minimum (rare) complexity guidelines as measured by some specified complexity metric: e.g., too many lines of code in module, cyclomatic complexity greater than 200, excessive Halstead length, too many tokens.

5215: Call nesting depth: violations of component (e.g., subroutine, subprogram, function) maximum nesting depth, or insufficient depth where dictated.

5216: Modularity and partition: Modularity and partition rules not followed: e.g., minimum and maximum size, object scope, functionally dictated partitions.

5217: Call nesting depth: violations of component (e.g., subroutine, subprogram, function) maximum nesting depth, or insufficient depth where dictated.

522x: Data definition, declarations: the form and/or location of data object declaration is not according to standards.

523x: Data access: violations of conventions governing how data objects of different kinds are to be accessed, wrong kind of object used: e.g., not using field-access macros, direct access instead of indirect, absolute reference instead of symbolic, access via register, etc.

524x: Calling and invoking: bugs in the manner in which other processing components are called, invoked, or communicated with: e.g., a direct subroutine call that should be indirect, violation of call and return sequence conventions.

526x: Mnemonics, label conventions: violations of the rules by which names are assigned to objects: e.g., program labels, subroutine and program names, data object names, file names.

5**27x: Format:** violations of conventions governing the overall format and appearance of the source code: indentation rules, pagination, headers, ID block, special markers.

528x: Comments: violations of conventions governing the use, placement, density, and format of comments. The content of comments is covered by 53xx, documentation.

53xx: DOCUMENTATION: bugs in the documentation associated with the code or the content of comments contained in the code.

531x: Incorrect: documentation statement is wrong.

532x: Inconsistent: documentation statement is inconsistent with itself or with other statements.

533x: Incomprehensible: documentation cannot be understood by a qualified reader.

534x: Incomplete: documentation is correct but important facts are missing.

535x: Missing: major parts of documentation are missing.

6xxx: INTEGRATION: bugs having to do with the integration of, and interfaces between, components. The components themselves are assumed to be correct.

61xx: INTERNAL INTERFACES: bugs related to the interfaces between communicating components with the program under test. The components are assumed to have passed their component level tests. In this context, direct or indirect transfer of data or control information via a memory object such as tables, dynamically allocated resources, or files, constitute an internal interface.

611x: Component invocation: bugs having to do with how software components are invoked. In this sense, a “component” can be a subroutine, function, macro, program, program segment, or any other sensible processing component. Note the use of “invoke” rather than “call” because there may be no actual call as such: e.g., a task order placed on a processing queue is an invocation in our sense, though (typically) not a call.

6111: No such component: invoked component does not exist.

6112: Wrong component: incorrect component invoked.

612x: Interface parameter, invocation: having to do with the parameter of the invocation, their number, order, type, location, values, etc.

6121: Wrong parameter: parameter of the invocation are incorrectly specified.

6122: Parameter type: incorrect invocation parameter type used.

6124: Parameter structure: structural details of parameter type used.

6125: Parameter value: value (numerical, Boolean, string) of the parameter is wrong.

6126: Parameter sequence: parameters of the invocation sequence in the wrong order, too many parameters, too few parameters.

613x: Component invocation return: having to do with the interpretation of parameters provided by the invoked component on return to the invoking component or on release of control to some other component. In this context, a record, a subroutine return sequence, or a file can qualify for this

category of bug. Note that the bugs included here are *not* bugs in the component that created the return data but in the receiving component's subsequent manipulation and interpretation of that data.

6131: Parameter identity: wrong return parameter accessed.

6132: Parameter type: wrong return parameter type used: that is, the component using the return data interprets a return parameter incorrectly as to type.

6134: Parameter structure: return parameter structure misinterpreted.

6136: Return sequence: sequence assumed for return parameter is incorrect.

614x: Initialization, state: invoked component not initialized or initialized to the wrong state or with incorrect data.

615x: Invocation in wrong place: the place or state in the invoking component at which the invoked component was invoked is wrong.

616x: Duplicate or spurious invocation: component should not have been invoked or has been invoked more often than necessary.

62xx: EXTERNAL INTERFACES AND TIMING: having to do with external interfaces, such as I/O devices and/or drivers, or other software not operating under the same control structure. Data passage by files or messages qualify for this bug category.

621x: Interrupts: bugs related to incorrect interrupt handling or setting up for interrupts: e.g., wrong handler invoked, failure to block or unblock interrupts.

622x: Devices and drivers: incorrect interface with devices or device drivers or incorrect interpretation of return status data.

6222: Device, driver, initialization or state: incorrect initialization of device or driver, failure to initialize, setting device to the wrong state.

6224: Device, driver, command bug: bug in the command issued to a device or driver.

6226: Device, driver, return/status misinterpretation: return status data from device or driver misinterpreted or ignored.

623x: I/O timing or throughput: bugs having to do with timing and data rates for external devices such as: not meeting specified timing requirements (too long or too short), forcing too much throughput, not accepting incoming data rates.

7xxx: SYSTEM AND SOFTWARE ARCHITECTURE: bugs that are not attributable to a component or to the interface between components but affect the entire software system or stem from architectural errors in the system.

71xx: OS bug: bugs related to the use of operating system facilities. Not to be confused with bugs in the operating system itself.

711x: Invocation, command: erroneous command given to operating system or OS facility incorrectly invoked.

712x: Return data, status misinterpretation: data returned from operating system or status information ignored or misinterpreted.

714x: Space: required memory (cache, disc, RAM) resource not available or requested in the wrong way.

72xx: Software architecture: architecture problems not elsewhere defined.

721x: Interlocks and semaphores: bugs in the use of interlock mechanisms and interprocess communication facilities. Not to be confused with bugs in these mechanisms themselves: e.g., failure to lock, failure to unlock, failure to set or reset semaphore, duplicate locking.

722x: Priority: bugs related to task priority: e.g., priority too low or too high, priority selected not allowed, priority conflicts.

723x: Transaction-flow control: where the path taken by a transaction through the system is controlled by an implicit or explicit transaction flow-control mechanism, these are bugs related to the definition of such flows. Note that all components and their interfaces could be correct but this kind of bug could still exist.

724x: Resource management and control: bugs related to the management of dynamically allocated shared resource objects: e.g., not returning a buffer block after use, not getting an object, failure to clean up an object after use, getting wrong kind of object, returning object to wrong pool.

725x: Recursive calls: bugs in the use of recursive invocation of software components or incorrect recursive invocation.

726x: Reentrance: bugs related to reentrance of program components: e.g., a reentrant component that should not be, a reentrant call that should be nonreentrant.

73xx: RECOVERY ACCOUNTABILITY: bugs related to the recovery of objects after the failure and to the accountability for objects despite failures.

74xx: PERFORMANCE: bugs related to the throughput-delay behavior of software under the assumption that all other aspects are correct.

741x: Throughput inadequate.

742x: Response time, delay: response time to incoming events too long at specified load or too short (rare), delay between outgoing events too long or too short.

7**43x: Insufficient users:** maximum specified number of simultaneous users or task cannot be accommodated at specified transaction delays.

748x: Performance parasites: any bug whose primary or only symptom is a performance degradation: e.g., the harmless but needless repetition of operations, fetching and returning more dynamic resources than needed.

75xx: INCORRECT DIAGNOSTIC, EXCEPTION: diagnostic or error message incorrect or misleading. Exception handler invoked is wrong.

76xx: PARTITIONS AND OVERLAYS: memory or virtual memory is incorrectly partitioned, overlay to wrong area, overlay or partition conflicts.

77xx: SYSGEN OR ENVIRONMENT: wrong operating system version, incorrect system generation, or other host environment problem.

8xxx: TEST DEFINTION OR EXCUTION BUGS: bugs in the definition, design, execution of tests or the data used in tests. These are as important as “real” bugs.

81xx: DESIGN BUGS: bugs in the design of tests.

811x: Requirements misunderstood: test and component are mismatched because test designer did not understand requirements.

812x: Incorrect outcome predicted: predicted outcome of test does not match required or actual outcome.

813x: Incorrect path predicted: outcome is correct but was achieved by the wrong predicted path. The test is only coincidentally correct.

814x: Test initialization: specified initial conditions for test are wrong.

815x: Test data structure or value: data objects used in tests or their values are wrong.

816x: Sequencing bug: the sequence in which tests are to be executed, relative to other tests or to test initialization, is wrong.

817x: Configuration: the hardware and/or software configuration and/or environment specified for the test is wrong.

818x: Verification method criteria: the method by which the outcome will be verified is incorrect or impossible.

82xx: EXECUTION BUGS: bugs in the execution of tests as contrasted with bugs in their design.

821x: Initialization: tested component not initialized to the right state or values.

822x: Keystroke or command: simple keystroke or button hit error.

823x: Database: database used to support the test was wrong.

824x: Configuration: configuration and/or environment specified for the test was not used during the run.

828x: Verification act: the act of verifying the outcome was incorrectly executed.

83xx: TEST DOCUMENTATION: documentation of test case or verification criteria is incorrect or misleading.

84xx: TEST CASE COMPLETENESS: cases required to achieve specified coverage criteria are missing.

Appendix B

GLOSSARY

Definitions for many of the technical terms used in the report are given below. An abbreviated indication of the reference from which the definition was taken is provided in square brackets.

610	IEEE 610-12
882C	MIL-STD-882C
1028	IEEE 1028
1058	IEEE 1058
1074	IEEE 1074
RADC	RADC 1977

Acceptability—A measure of how closely the computer program meets the true needs of the user [RADC].

Accessibility—the extent that software facilitates the selective use of its components [RADC].

Augmentability—the extent that software easily accommodates expansions in data storage requirements or component computational functions [RADC].

Accountability—the extent that code usage can be measured [RADC].

Accuracy—(1) A qualitative assessment of correctness, or freedom from error [610]. (2) A quantitative measure of the magnitude of error [610]. (3) A measure of the quality of freedom from error, degree of exactness possessed by an approximation or measurement [RADC].

Activity—(1) A group of related tasks [IEEE 1074]. (2) A major unit of work to be completed in achieving the objectives of a software project. An activity has precise starting and ending dates, incorporates a set of tasks to be completed, consumes resources and results in work products [1058].

Adaptability—The ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed [610].

Availability—(1) The degree to which a system or component is operational and accessible when required for use [610]. (2) The fraction of total time during which the system can support critical functions [RADC]. (3) The probability that a system is operating satisfactorily at any point in time, when used under stated conditions [RADC].

Clarity—(1) The ease with which the program (and its documentation) can be understood by humans [RADC]. (2) The extent to which a document contains enough information for a reader to determine its objectives, assumptions, constraints, inputs, outputs, components, and status [RADC].

Completeness—(1) The attributes of software that provide full implementation of the functions required [RADC]. (2) The extent to which software fulfills overall mission satisfaction [RADC]. (3) The extent that all of the software's parts are present and each of its parts are fully developed [RADC].

Consistency—The degree of uniformity, standardization, and freedom from contradiction among the documents or parts of a system or component [610].

Convertibility—The degree of success anticipated in readying people, machines, and procedures to support the system [RADC].

Cost—Includes not only development cost, but also the costs of maintenance, training, documentation, etc., on the entire life cycle of the program [RADC].

Correctness—(1) The degree to which a system or component is free from faults in its specification, design and implementation [610]. (2) The degree to which software, documentation, or other items meet specified requirements [610]. (3) The degree to which software, documentation or other items meet user needs and expectations, whether specified or not [610].

Extendibility—The ease with which a system or component can be modified to increase its storage or functional capacity [610].

Generality—a measure of the scope of the functions that a program performs [RADC].

Inexpensiveness—see Cost.

Integrity—(1) The degree to which a system or component prevents unauthorized access to, or modification of, computer programs or data [610]. (2) A measure of the degree of protection the computer program offers against unauthorized access and loss due to controllable events [RADC]. (3) The ability of software to prevent purposeful or accidental damage to the data or software [RADC].

Interface—(1) A shared boundary across which information is passed [610]. (2) A hardware or software component that connects two or more components for the purpose of passing information from one to the other [610].

Interoperability—how quickly and easily one software system can be coupled to another [RADC].

Maintainability—(1) The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment [610]. (2) The probability that a failed system will be restored to operable conditions within a specified time [RADC].

Manageability—the degree to which a system lends itself to efficient administration of its components [RADC].

Modifiability—(1) A measure of the cost of changing or extending a program [RADC]. (2) The extent to which a program facilitates the incorporation of changes, once the nature of the desired change has been determined [RADC].

Modularity—(1) The degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components [610]. (2) The ability to combine arbitrary program modules into larger modules without knowledge of the construction of the modules [RADC]. (3) A formal way of dividing a program into a number of sub-units each having a well defined function and relationship to the rest of the program [RADC].

Non-complexity—see Simplicity.

Performance—(1) The degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage [610]. (2) The effectiveness with which resources of the host system are utilized toward meeting the objective of the software system [RADC].

Portability—The ease with which a system or component can be transferred from one hardware or software environment to another [610].

Precision—(1) The degree of exactness or discrimination with which a quantity is stated [610]. (2) The degree to which calculated results reflect theoretical values [RADC].

Reliability—(1) The ability of a system or component to perform its required functions under stated conditions for a specified period of time [610]. (2) The probability that a software system will operate without failure for at least a given period of time when used under stated conditions [RADC]. (3) The probability that a software fault does not occur during a specified time interval (or specified number of software operational cycles) which causes deviation from required output by more than specified tolerances, in a specific environment [RADC].

Reparability—The probability that a failed system will be restored to operable condition within a specified active repair time when maintenance is done under specified conditions [RADC].

Requirement—(1) A condition or capability needed by a user to solve a problem or achieve an objective [610]. (2) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification or other formally imposed documents [610].

Reusability—The degree to which a software module or other work product can be used in more than one computer program or software system [610].

Review—An evaluation of software elements or project status to ascertain discrepancies from planned results and to recommend improvement [1028].

Robustness—(1) The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions [610]. (2) The quality of a program that determines its ability to continue to perform despite some violation of the assumptions in its specification [RADC].

Safety—Freedom from those conditions that can cause death, injury, occupational illness or damage to or loss of equipment or property, or damage to the environment [882C].

Security—(1) A measure of the probability that one system user can accidentally or intentionally reference or destroy data that is the property of another user or interfere with the operation of the system [RADC]. (2) The extent to which access to software, data and facilities can be controlled [RADC].

Self-Descriptiveness—The degree to which a system or component contains enough information to explain its objectives and properties [610].

Serviceability—The degree of ease or difficulty with which a system can be repaired [RADC].

Simplicity—The degree to which a system or component has a design and implementation that is straightforward and easy to understand [610].

Software products—(1) The complete set of computer programs, procedures and possibly associated documentation and data designated for delivery to a user [610]. (2) Any of the individual items in (1) [610].

Structuredness—(1) The ability to combine arbitrary program modules into larger modules without knowledge of the construction of the modules [RADC]. (2) The extent to which a system possesses a definite pattern of organization of its independent parts [RADC]. (3) A formal way of dividing a program into a number of sub-units each having a well defined function and relationship to the rest of the program [RADC].

Task—The smallest unit of work subject to management accountability. A task is a well-defined work assignment for one or more project members. [1074]

Testability—(1) The degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met [610]. (2) The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met [610].

Understandability—(1) The extent to which the purpose of the product is clear to the evaluator [RADC]. (2) The ease with which an implementation can be understood [RADC].

Uniformity—a module should be usable uniformly [RADC].

Usability—(1) The ease with which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component [610]. (2) The ease of operation from the human viewpoint, covering both human engineering and ease of transition from current operation [RADC].

User Friendliness—the degree of ease of use of a computer system, device, program, or document. See User Friendly in [610].

Validation—The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements [610].

Validity—The degree to which software implements the user's specifications [RADC].

Verification—The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase [610].

Verification and Validation—The process of determining whether the requirements for a system or component are complete and correct, the products of each development phase fulfill the requirements or conditions imposed by the previous phase, and the final system or component complies with specified requirements [610].